

Wykład OpenMP część 2

Programowanie równoległe w systemie z pamięcią wspólną.

1. Synchronizacja wątków
2. Wątek Master
3. Klauzule
4. Przykład użycia funkcji OpenMP

Dyrektywy synchronizacji pozwalają zaprowadzić porządek w dostępie do zasobów współdzielonych.

Dysponujemy w OpenMP

- barierą
- dyrektywą porządkowania wątków
- sekcją krytyczną
- operacjami atomowymi

Bariera

```
#pragma omp barrier
```

W miejscu tak oznaczonym wątki czekają na siebie i przechodzą do dalszej części kodu tylko wspólnie.

Wszystkie wątki z team-u muszą mieć szanse na dotarcie do bariery.

Kolejność barier i obszarów rozdziału pracy musi być taka sama dla wszystkich wątków.

Dyrektywa porządkowania wątków

```
#pragma omp ordered
```

W obszarze zrównoleglenia pracy możliwe jest oznaczenie regionu, który ma być wykonywany w naturalnej kolejności (tzn. tak jakby kod był sekwencyjny).

Ważne: program nadal działa wielowątkowo i praca nadal jest dzielona pomiędzy wątki.

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

using namespace std;

int main ( void ) {
    int i, j, sum;
#pragma omp parallel for private(i,j,sum)
    for ( i = 1; i < 20; i++ ) {
        sum = 0;
        for ( j = 0; j <= i; j++ )
            sum += j;
        cout << "To ja watek nr " << omp_get_thread_num()
            << " suma od 0 do " << i << " wynosi: "
            << sum << endl;
    } // for (i)
    return 0;
} // main()
```

```
> export OMP_NUM_THREADS=4
> ./a.out
To ja watek nr 3 suma od 0 do 16 wynosi: 136
To ja watek nr 2 suma od 0 do 11 wynosi: 66
To ja watek nr 2 suma od 0 do 12 wynosi: 78
To ja watek nr 2 suma od 0 do 13 wynosi: 91
To ja watek nr 2 suma od 0 do 14 wynosi: 105
To ja watek nr 2 suma od 0 do 15 wynosi: 120
To ja watek nr 1 suma od 0 do 6 wynosi: 21
To ja watek nr 1 suma od 0 do 7 wynosi: 28
To ja watek nr 1 suma od 0 do 8 wynosi: 36
To ja watek nr 1 suma od 0 do 9 wynosi: 45
To ja watek nr 1 suma od 0 do 10 wynosi: 55
To ja watek nr 3 suma od 0 do 17 wynosi: 153
To ja watek nr 3 suma od 0 do 18 wynosi: 171
To ja watek nr 3 suma od 0 do 19 wynosi: 190
To ja watek nr 0 suma od 0 do 1 wynosi: 1
To ja watek nr 0 suma od 0 do 2 wynosi: 3
To ja watek nr 0 suma od 0 do 3 wynosi: 6
To ja watek nr 0 suma od 0 do 4 wynosi: 10
To ja watek nr 0 suma od 0 do 5 wynosi: 15
```

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

using namespace std;

int main ( void ) {
    int i, j, sum;
#pragma omp parallel for private(i,j,sum) ordered
    for ( i = 1; i < 20; i++ ) {
        sum = 0;
        for ( j = 0; j <= i; j++ )
            sum += j;
#pragma omp ordered
        cout << "To ja watek nr " << omp_get_thread_num()
            << " suma od 0 do " << i << " wynosi: "
            << sum << endl;
    } // for (i)
    return 0;
} // main()
```

```
> export OMP_NUM_THREADS=4
> ./a.out
To ja watek nr 0 suma od 0 do 1 wynosi: 1
To ja watek nr 0 suma od 0 do 2 wynosi: 3
To ja watek nr 0 suma od 0 do 3 wynosi: 6
To ja watek nr 0 suma od 0 do 4 wynosi: 10
To ja watek nr 0 suma od 0 do 5 wynosi: 15
To ja watek nr 1 suma od 0 do 6 wynosi: 21
To ja watek nr 1 suma od 0 do 7 wynosi: 28
To ja watek nr 1 suma od 0 do 8 wynosi: 36
To ja watek nr 1 suma od 0 do 9 wynosi: 45
To ja watek nr 1 suma od 0 do 10 wynosi: 55
To ja watek nr 2 suma od 0 do 11 wynosi: 66
To ja watek nr 2 suma od 0 do 12 wynosi: 78
To ja watek nr 2 suma od 0 do 13 wynosi: 91
To ja watek nr 2 suma od 0 do 14 wynosi: 105
To ja watek nr 2 suma od 0 do 15 wynosi: 120
To ja watek nr 3 suma od 0 do 16 wynosi: 136
To ja watek nr 3 suma od 0 do 17 wynosi: 153
To ja watek nr 3 suma od 0 do 18 wynosi: 171
To ja watek nr 3 suma od 0 do 19 wynosi: 190
```

Sekcja krytyczna

```
#pragma omp critical [(nazwa sekcji)]
```

Tylko jeden wątek może realizować kod zawarty w takiej sekcji.

Dzięki możliwości nadawania sekcjom nazw można stworzyć ich kilka i niezależnie chronić do nich dostęp.

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

using namespace std;

int main ( void ) {
    int i, j, sum;
    int sum_sum = 0;
#pragma omp parallel for private(i,j,sum) shared(sum_sum)
    for ( i = 1; i < 20; i++ ) {
        sum = 0;
        for ( j = 0; j <= i; j++ )
            sum += j;
#pragma omp critical
        {
            sum_sum += sum;
            cout << "To ja watek nr " << omp_get_thread_num()
                << " suma od 0 do " << i << " wynosi: "
                << sum << " aktualizuje sum_sum " << endl;
        }
    } // for i
    cout << "Sum_sum = " << sum_sum << endl;
```

To trzeba
chronić

Wydruk
także!

```
> export OMP_NUM_THREADS=4
> ./a.out
To ja watek nr 3 suma od 0 do 16 wynosi: 136 aktualizuje sum_sum
To ja watek nr 1 suma od 0 do 6 wynosi: 21 aktualizuje sum_sum
To ja watek nr 1 suma od 0 do 7 wynosi: 28 aktualizuje sum_sum
To ja watek nr 1 suma od 0 do 8 wynosi: 36 aktualizuje sum_sum
To ja watek nr 1 suma od 0 do 9 wynosi: 45 aktualizuje sum_sum
To ja watek nr 1 suma od 0 do 10 wynosi: 55 aktualizuje sum_sum
To ja watek nr 2 suma od 0 do 11 wynosi: 66 aktualizuje sum_sum
To ja watek nr 2 suma od 0 do 12 wynosi: 78 aktualizuje sum_sum
To ja watek nr 2 suma od 0 do 13 wynosi: 91 aktualizuje sum_sum
To ja watek nr 2 suma od 0 do 14 wynosi: 105 aktualizuje sum_sum
To ja watek nr 2 suma od 0 do 15 wynosi: 120 aktualizuje sum_sum
To ja watek nr 3 suma od 0 do 17 wynosi: 153 aktualizuje sum_sum
To ja watek nr 3 suma od 0 do 18 wynosi: 171 aktualizuje sum_sum
To ja watek nr 3 suma od 0 do 19 wynosi: 190 aktualizuje sum_sum
To ja watek nr 0 suma od 0 do 1 wynosi: 1 aktualizuje sum_sum
To ja watek nr 0 suma od 0 do 2 wynosi: 3 aktualizuje sum_sum
To ja watek nr 0 suma od 0 do 3 wynosi: 6 aktualizuje sum_sum
To ja watek nr 0 suma od 0 do 4 wynosi: 10 aktualizuje sum_sum
To ja watek nr 0 suma od 0 do 5 wynosi: 15 aktualizuje sum_sum
Sum_sum = 1330
```

```
do
{
    sum_sum = 0;
#pragma omp parallel for private(i, j, sum) shared( sum_sum )
    for ( i = 1; i < 20; i++ )
    {
        sum = 0;
        for ( j = 0; j <= i; j++ )
            sum += j;
//#pragma omp critical
        {
            sum_sum += sum;
        }
    }
}
while ( sum_sum == 1330 );

cout << "Sum_sum = " << sum_sum << endl;
```

Powyższy program to mały eksperyment – chcemy sprawdzić jak długo trzeba czekać aby pojawił się błąd obliczeniowy w kodzie bez zabezpieczeń dostępu do współdzielonej zmiennej sum_sum.

```
[root@ciserv ~]# time ./a.out
Sum_sum = 1093
real 0m0.024s
user 0m0.006s
sys 0m0.017s

[root@ciserv ~]# time ./a.out
Sum_sum = 1210
real 0m0.013s
user 0m0.002s
sys 0m0.013s

[root@ciserv ~]# time ./a.out
Sum_sum = 1285
real 0m0.277s
user 0m0.116s
sys 0m0.258s

[root@ciserv ~]# time ./a.out
Sum_sum = 1299
real 0m0.008s
user 0m0.003s
sys 0m0.005s
```

Program działał na starym laptopie z jednordzeniowym procesorem Pentium 4 Mobile 1.7GHz...

Operacje atomowe

```
#pragma omp atomic
```

Tylko jeden wątek może wykonywać jednocześnie taką operację. Kompilator używa wsparcia w sprzęcie. Ta dyrektywa odnosi się tylko do jednej instrukcji po niej

```
#pragma omp atomic  
    sum_sum += sum;
```



```
#pragma omp atomic  
    sum_sum += sum();
```



Wspierane operacje to : +, *, -, /, &, ^, |, <<, >>.

Operacja wykonywana wyłącznie przez wątek Master

```
#pragma omp master
```

Uwaga: tu nie ma barier ani na wejściu ani na wyjściu !!! Ale zawsze można dodać ręcznie .

Klauzula warunkowa

```
#pragma omp parallel if( warunek )
```

Jeśli warunek jest spełniony uaktywniane są wątki, jeśli nie to działa tylko wątek Master

```
#pragma omp parallel for if ( max > 1000 )
for ( i = 0; i < max; i++ )
{
    // coś
}
```

Tworzenie wątków nie zawsze się opłaca. Chcemy zmierzyć czas narzutu związanego z pracą OpenMP:

```
int main ( void ) {  
  
    int i,reps = 500000 ;  
    struct timeval ti;  
  
    int *sum = new int [ 400 ];  
    for ( int i = 0; i < 400; i++ ) sum[ i ] = 0;  
  
    gettimeofday( &ti, NULL );  
  
    for ( int r = 0; r < reps; r++ ) {  
#pragma omp parallel for  
        for ( int i = 0; i < 400; i++ ) sum[ i ] += 1;  
    }  
  
    show( &ti );  
    return 0;  
}
```

Bardzo dużo razy wykonujemy równolegle prostą pętle

```
for ( int r = 0; r < reps; r++ )  
{  
#pragma omp parallel for  
    for ( int i = 0; i < 400; i++ ) sum[ i ] += 1;  
}
```

reps	Bez OpenMP	OpenMP	Różnica / reps
500 000	0.167s	12.772s	25μs
1 000 000	0.312s	24.875s	24.5μs
2 000 000	0.498s	50.678s	25.1μs
5 000 000	1.192s	125.76s	24.9μs

Klauzula ustawienia ilości wątków. Działa tylko dla omp parallel.

```
#pragma omp parallel num_threads( int )\n\nint i;\nint max = 20;\n#pragma omp parallel for private(i) shared( max ) \\\n    num_threads( (int)( max / 2 ) )\nfor ( i = 1; i < max; i++ )\n{\n    cout << "To ja watek nr "\n        << omp_get_thread_num() << endl;\n}
```

Ile wątków będzie realizować pętle for() ?

W OpenMP istnieje kilka sposobów ustawienia ilości wątków

1. Zmienna systemowa: `OMP_NUM_THREADS`
2. Funkcja biblioteczna: `omp_set_num_threads(int)` - ta funkcja nie może być wykonywana w obszarze równoległym!
3. Klauzula `num_threads(int)`

Generalna zasada: im bliżej bloku równoległego następuje ustawienie ilości wątków tym jest ono ważniejsze

```
void show( void ) {  
#pragma omp for  
for ( int i = 0; i < 6; i++ )  
    cout << "To ja watek " << omp_get_thread_num() << " z "  
        << omp_get_num_threads() << " i = " << i << endl;  
}  
  
int main ( void ) {  
  
cout << "Maks liczba watkow " << omp_get_max_threads() << endl;  
  
omp_set_num_threads( 10 );  
cout << "Maks liczba watkow " << omp_get_max_threads() << endl;  
  
#pragma omp parallel num_threads( 3 )  
{  
#pragma omp master  
    cout << "Uzywana liczba w " << omp_get_num_threads() << endl;  
#pragma omp barrier  
    show();  
}  
    show();  
}
```

```
> env OMP_NUM_THREADS=5 ./a.out
Maksymalna liczba wątków 5
Maksymalna liczba wątków 10
Używana liczba wątków 3
To ja wątek 1 z 3 i = 2
To ja wątek 1 z 3 i = 3
To ja wątek 2 z 3 i = 4
To ja wątek 2 z 3 i = 5
To ja wątek 0 z 3 i = 0
To ja wątek 0 z 3 i = 1
To ja wątek 0 z 1 i = 0
To ja wątek 0 z 1 i = 1
To ja wątek 0 z 1 i = 2
To ja wątek 0 z 1 i = 3
To ja wątek 0 z 1 i = 4
To ja wątek 0 z 1 i = 5
```

Wynik po wywołaniu:
omp_set_num_threads(10);

#pragma omp parallel
num_threads(3)

Funkcja show() wywołana z
bloku parallel

Funkcja show() wywołana po
zakończeniu bloku parallel

```
void show( void ) {
if ( omp_in_parallel() ) {
    cout << "Och, wywolano mnie z aktywnego regionu row" << endl;
}
else {
    cout << "Co za nuda, jeden watek i tyle" << endl;
    if ( omp_get_num_procs() == 1 ) {
        cout << "Juz rozumiem masz tylko jeden procesor" << endl;
    }
}
#pragma omp for
for ( int i = 0; i < 6; i++ )
    cout << "To ja watek " << omp_get_thread_num() << " z "
        << omp_get_num_threads() << " i = " << i << endl;
}
int main ( void ) {
cout << "Maks liczba watkow " << omp_get_max_threads() << endl;
omp_set_num_threads( 10 );
cout << "Maks liczba watkow " << omp_get_max_threads() << endl;
#pragma omp parallel num_threads( 3 )
{
#pragma omp master
    cout << "Uzywana liczba w " << omp_get_num_threads() << endl;
#pragma omp barrier
    show();
}
show();
}
```

```
> env OMP_NUM_THREADS=5 ./a.out
Maksymalna liczba wątków 5
Maksymalna liczba wątków 10
Używana liczba wątków 3
Och, wywołano mnie z aktywnego regionu
To ja wątek 1 z 3 i = 2
To ja wątek 1 z 3 i = 3
Och, wywołano mnie z aktywnego regionu
To ja wątek 2 z 3 i = 4
To ja wątek 2 z 3 i = 5
Och, wywołano mnie z aktywnego regionu
To ja wątek 0 z 3 i = 0
To ja wątek 0 z 3 i = 1
Co za nuda, jeden wątek i tyle
Juz rozumiem masz tylko jeden procesor
To ja wątek 0 z 1 i = 0
To ja wątek 0 z 1 i = 1
To ja wątek 0 z 1 i = 2
To ja wątek 0 z 1 i = 3
To ja wątek 0 z 1 i = 4
To ja wątek 0 z 1 i = 5
```

Wynik po wywołaniu:
omp_set_num_threads(10);

#pragma omp parallel
num_threads(3)

rownolegle
Funkcja show()
wywołana z bloku parallel
rownolegle

Funkcja show() wywołana po
zakończeniu bloku parallel

```
@baribal:~> icc -openmp openmp.cpp  
openmp.cpp(18) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.  
openmp.cpp(31) : (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.  
openmp.cpp(18) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.  
openmp.cpp(18) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
```

```
@baribal:~> ./a.out  
Maksymalna liczba wątków 256  
Maksymalna liczba wątków 10  
Używana liczba wątków 3  
Och, wywołano mnie z aktywnego regionu równoległego  
Och, wywołano mnie z aktywnego regionu równoległego  
Och, wywołano mnie z aktywnego regionu równoległego
```

```
To ja wątek To ja wątek 1 z 32 z i = 2  
To ja wątek 1 z 3 i = 3  
3 i = 4  
To ja wątek 2 z 3 i = 5  
To ja wątek 0 z 3 i = 0  
To ja wątek 0 z 3 i = 1  
Co za nuda, jeden wątek i tyle  
To ja wątek 0 z 1 i = 0  
To ja wątek 0 z 1 i = 1  
To ja wątek 0 z 1 i = 2  
To ja wątek 0 z 1 i = 3  
To ja wątek 0 z 1 i = 4  
To ja wątek 0 z 1 i = 5
```

Funkcja show()
wywołana po
zakończeniu
bloku parallel

Tak, ta maszyna ma
256 procesorów !
Tyle także zwraca funkcja
omp_get_num_procs()



Klauzula redukcji

```
#pragma omp for reduction( operator : zmienne )
```

Zamiast ręcznego tworzenia sekcji krytycznych zawsze używaj **reduction** !!!

Zmienne wskazane w klauzuli **reduction** traktowane są automatycznie jako współdzielone.

Początkowa wartość używanej do redukcji zmiennej jest aktualizowana, a nie nadpisywana, czyli zmienną należy zainicjować.

Ale uwaga na dokładność operacji!!! Kolejność może być inna niż w przypadku kodu sekwencyjnego.

Klauzula redukcji

```
#pragma omp for reduction( operator : zmienne )
```

KOLEJNOŚĆ WYKONYWANIA OPERACJI NIE JEST
GWARANTOWANA !!!!

10.00	1.001	1.009
+ 1.001	+1.009	+10.00
+ 1.009	+10.00	1.001
=12.00	=12.01	=12.00

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

using namespace std;

int main ( void ) {
    int i, j, sum;
    int sum_sum = 0;
#pragma omp parallel for private(i,j,sum) \
    reduction(+ : sum_sum)
    for ( i = 1; i < 20; i++ ) {
        sum = 0;
        for ( j = 0; j <= i; j++ )
            sum += j;
        sum_sum += sum; // to będzie chronione !!!
    }
}
```

Klauzula kopiowania zmiennych prywatnych wątku

```
#pragma omp single copyprivate( lista )
```

Prywatne zmienne wątku realizującego sekcję **single** są kopiowane do odpowiednich zmiennych prywatnych innych wątków

Klauzula **nowait** jest zabroniona !

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

using namespace std;

int main ( void ) {
    int i = 0;

    #pragma omp parallel private( i )
    #pragma omp critical
    {
        cout << "Watek nr " << omp_get_thread_num() << " i = "
            << i << endl;
    }
}
```

```
> env OMP_NUM_THREADS=5 ./a.out
Watek nr 4 i = 148616848
Watek nr 2 i = 148616848
Watek nr 1 i = 148616848
Watek nr 3 i = 148616848
Watek nr 0 i = -1076225676
```

```
#pragma omp parallel private( i )
{
#pragma omp single
{
    cout << "To wykonuje tylko ja, watek nr. " <<
        omp_get_thread_num() << endl;
    i = 100 / 5;
}

#pragma omp critical
{
    cout << "Watek nr " << omp_get_thread_num() << " i = "
        << i << endl;
}
}

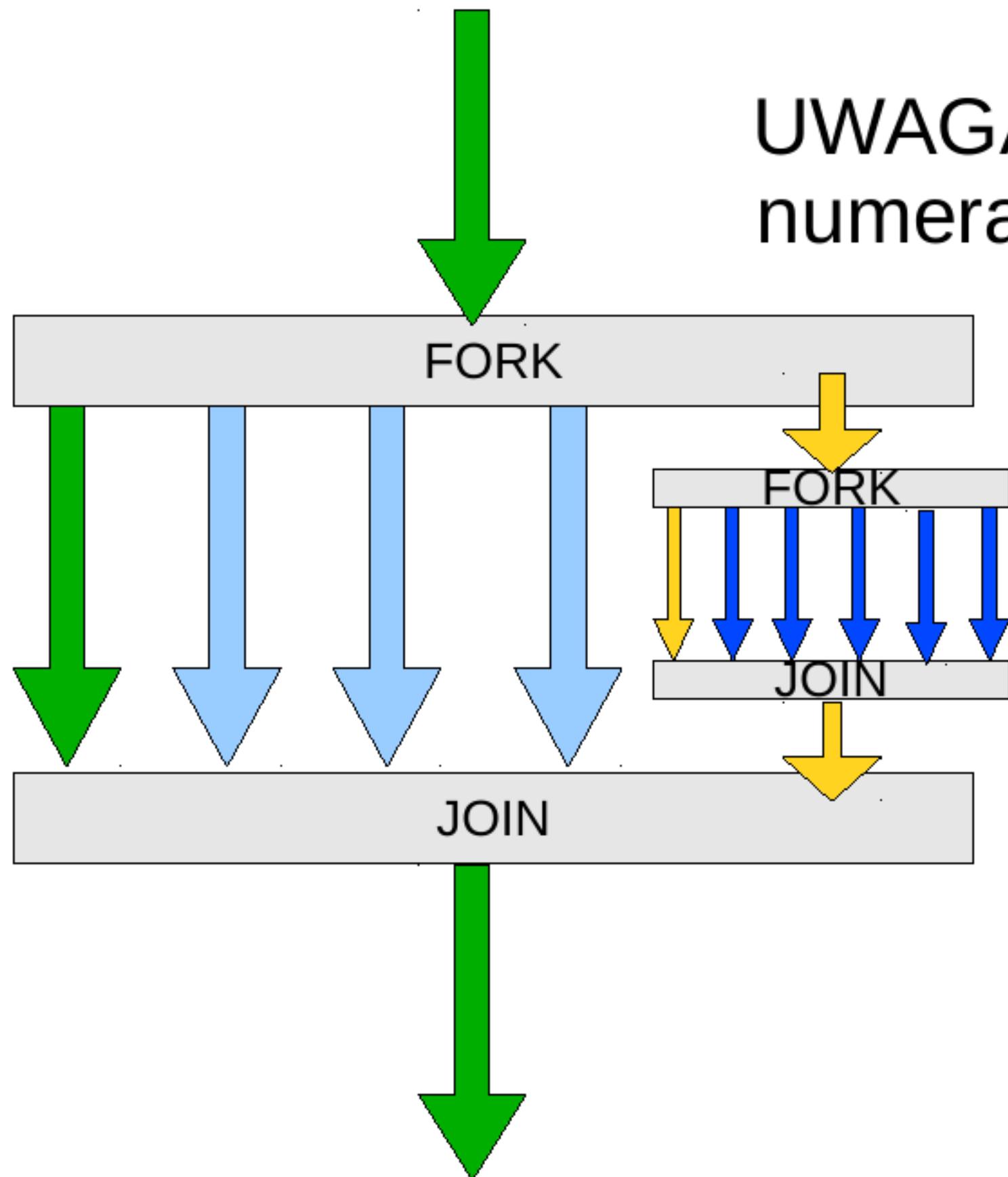
> env OMP_NUM_THREADS=5 ./a.out
To wykonuje tylko ja, watek nr. 4
Watek nr 4 i = 20
Watek nr 1 i = 151721616
Watek nr 2 i = 151721616
Watek nr 3 i = 151721616
Watek nr 0 i = -1077237628
```

```
#pragma omp parallel private( i )
{
#pragma omp single copyprivate( i )
{
    cout << "To wykonuje tylko ja, watek nr. " <<
        omp_get_thread_num() << endl;
    i = 100 / 5;
}

#pragma omp critical
{
    cout << "Watek nr " << omp_get_thread_num() << " i = "
        << i << endl;
}
}

> env OMP_NUM_THREADS=5 ./a.out
To wykonuje tylko ja, watek nr. 4
Watek nr 3 i = 20
Watek nr 1 i = 20
Watek nr 2 i = 20
Watek nr 4 i = 20
Watek nr 0 i = 20
```

Zrównoleglenie zagnieżdżone (nested parallelism)



UWAGA na zmianę w numeracji wątków !

```
omp_set_nested( true );

#pragma omp parallel sections
{
#pragma omp section
{
    int n = omp_get_thread_num();
    cout << "To ja watek " << n << " ide spac..." << endl;
    sleep( 10 );
    cout << "To ja watek " << n << ", ktos mnie obudzil" << endl;
} // section

#pragma omp section
{
    int n = omp_get_thread_num();
    cout << " To ja inny watek " << n << " start" << endl;

    if ( omp_get_nested() )
    {
        cout << " Dziala zagniezdzone zrownoleglanie, no to zobaczymy" << endl;

#pragma omp parallel for
        for ( int i = 0; i < 5; i++ )
            cout << "      Dziala teraz watek numer " << omp_get_thread_num() << endl;
    }

    cout << " To ja inny watek " << n << " stop" << endl;
} // section
} // sections
```

```
$ env OMP_NUM_THREADS=5 ./a.out
```

To ja watek 4 ide spac...

To ja inny watek 2 start

Dziala zagniezdzzone zrownoleglanie, no to zobaczymy

Dziala teraz watek numer 4

Dziala teraz watek numer 2

Dziala teraz watek numer 1

Dziala teraz watek numer 3

Dziala teraz watek numer 0

To ja inny watek 2 stop

To ja watek 4, ktos mnie obudzil

Wymuszenie konsystencji współdzielonych danych

```
#pragma omp flush( lista zmiennych )
```

W OpenMP wątki synchronizują współdzielone zmienne w punktach synchronizacji – do chwili osiągnięcia takiego punktu wątki mają prawo do przechowywania danych w pamięci podręcznej.

Punkty synchronizacji występują niejawnie na końcu bloków:

- * omp parallel (*),
- *omp critical (*),
- *ordered (*),
- * omp for,
- * omp sections,
- * omp single,

oraz wewnątrz obszaru

- * omp barrier

W przypadku konstrukcji podziału pracy punkt synchronizacji nie występuje gdy użyto klauzuli nowait .

(*) Do synchronizacji dochodzi również na początku bloku.

Wymuszenie konsystencji współdzielonych danych

```
#pragma omp flush( lista zmiennych )
```

Niejawnie **flush** jest wywoływany na granicy sekcji krytycznej.

Dlaczego uważać na operację **flush** ?



Co zrobić ze zmiennymi globalnymi ?

```
int zmienna_globalna = 5;

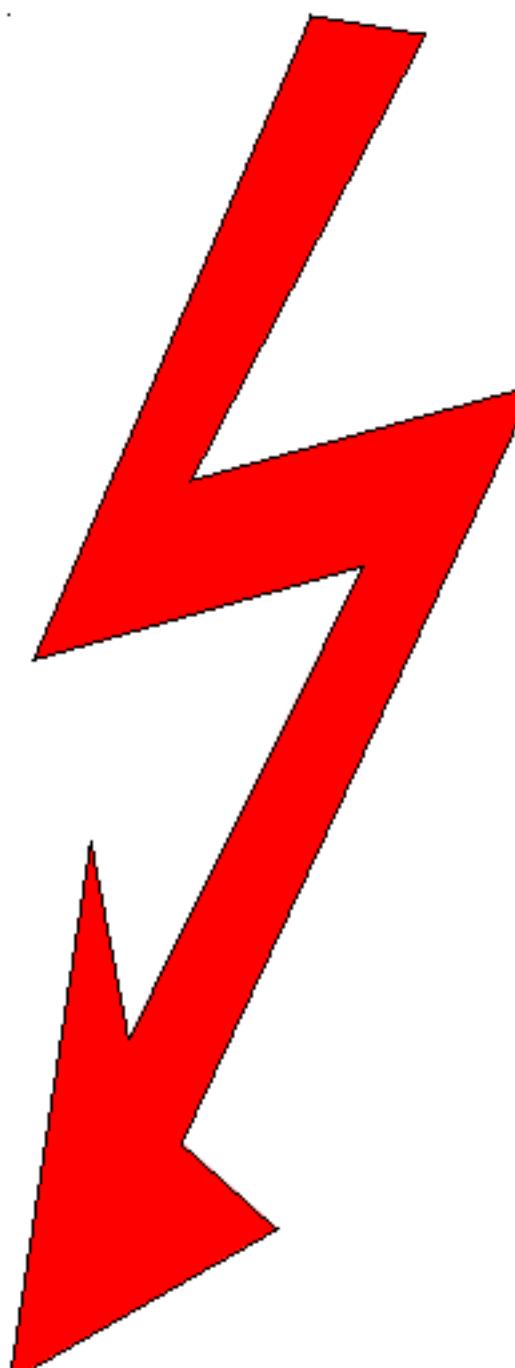
void show_zg( int n )
{
#pragma omp critical
    cout << "Watek " << n << " : Zmienna globalna zawiera : "
        << zmienna_globalna << endl;
}

int main ( void ) {
#pragma omp parallel for
    for ( int i = 1; i < 10; i++ ) {
        int n = omp_get_thread_num();
        zmienna_globalna = 10 + n;
        show_zg( n );
    }
    return 0;
}
```

Co zrobić ze zmiennymi globalnymi ?

```
>env OMP_NUM_THREADS=5 ./a.out
```

```
Watek 4 : Zmienna globalna zawiera : 14  
Watek 2 : Zmienna globalna zawiera : 13  
Watek 2 : Zmienna globalna zawiera : 12  
Watek 1 : Zmienna globalna zawiera : 12  
Watek 1 : Zmienna globalna zawiera : 11  
Watek 3 : Zmienna globalna zawiera : 11  
Watek 3 : Zmienna globalna zawiera : 13  
Watek 0 : Zmienna globalna zawiera : 10  
Watek 0 : Zmienna globalna zawiera : 10
```



Zmienne prywatne wątku na poziomie całego programu

```
#pragma omp threadprivate( lista zmiennych )
```

Wymienione zmienne stają się prywatne dla każdego wątku a ich zasięg to cały program.

Co zrobić ze zmiennymi globalnymi ?

```
int zmienna_globalna = 5;  
#pragma omp threadprivate( zmienna_globalna )  
  
void show_zg( int n )  
{  
#pragma omp critical  
    cout << "Watek " << n << " : Zmienna globalna zawiera : "  
        << zmienna_globalna << endl;  
}  
  
int main ( void ) {  
#pragma omp parallel for  
    for ( int i = 1; i < 10; i++ ) {  
        int n = omp_get_thread_num();  
        zmienna_globalna = 10 + n;  
        show_zg( n );  
    }  
    return 0;  
}
```

Co zrobić ze zmiennymi globalnymi ?

```
>env OMP_NUM_THREADS=5 ./a.out
```

```
Watek 4 : Zmienna globalna zawiera : 14
Watek 2 : Zmienna globalna zawiera : 12
Watek 2 : Zmienna globalna zawiera : 12
Watek 1 : Zmienna globalna zawiera : 11
Watek 1 : Zmienna globalna zawiera : 11
Watek 3 : Zmienna globalna zawiera : 13
Watek 3 : Zmienna globalna zawiera : 13
Watek 0 : Zmienna globalna zawiera : 10
Watek 0 : Zmienna globalna zawiera : 10
```

```
int *wsk_do_tablicy; // zmienna globalna

#pragma omp threadprivate( wsk_do_tablicy )

int suma( int len ) { // funkcja uzywa zmiennej globalnej
    int sum = 0;
    for ( int i = 0; i < len; i++ )
        sum += wsk_do_tablicy[ i ];
}

[...]

#pragma omp parallel for
for( int j = 0; j < calcs; j++ )
{
    wsk_do_tablicy = new int [ rozmiar ];
    [...]
    int l_suma = suma( rozmiar );
    delete[] wsk_do_tablicy;
}
```

Klauzula kopiowania zmiennych prywatnych wątku Master

```
#pragma omp parallel copyin( lista )
```

Prywatne zmienne wątku Master są kopiowane do odpowiednich zmiennych prywatnych innych wątków

```
int zmienna_globalna = 5;  
#pragma omp threadprivate( zmienna_globalna )  
  
void show_zg( int n )  
{  
#pragma omp critical  
    cout << "Watek " << n << " : Zmienna globalna zawiera : "  
        << zmienna_globalna << endl;  
}  
  
int main ( void ) {  
    show_zg(-1);  
    zmienna_globalna = 6;  
    show_zg(-2);  
  
#pragma omp parallel for  
    for ( int i = 1; i < 10; i++ ) {  
        int n = omp_get_thread_num();  
        show_zg( n );  
    }  
    return 0;  
}
```

} Część sekwencyjna

} Część równoległa

```
$ env OMP_NUM_THREADS=5 ./a.out  
Watek -1 : Zmienna globalna zawiera : 5  
Watek -2 : Zmienna globalna zawiera : 6  
Watek 4 : Zmienna globalna zawiera : 5  
Watek 2 : Zmienna globalna zawiera : 5  
Watek 2 : Zmienna globalna zawiera : 5  
Watek 1 : Zmienna globalna zawiera : 5  
Watek 1 : Zmienna globalna zawiera : 5  
Watek 3 : Zmienna globalna zawiera : 5  
Watek 3 : Zmienna globalna zawiera : 5  
Watek 0 : Zmienna globalna zawiera : 6  
Watek 0 : Zmienna globalna zawiera : 6
```



```
int zmienna_globalna = 5;
#pragma omp threadprivate( zmienna_globalna )

void show_zg( int n )
{
#pragma omp critical
    cout << "Watek " << n << " : Zmienna globalna zawiera : "
        << zmienna_globalna << endl;
}

int main ( void ) {
    show_zg(-1);
    zmienna_globalna = 6;
    show_zg(-2);

#pragma omp parallel for copyin( zmienna_globalna )
    for ( int i = 1; i < 10; i++ )          {
        int n = omp_get_thread_num();
        show_zg( n );
    }
    return 0;
}
```

```
$ env OMP_NUM_THREADS=5 ./a.out  
Watek -1 : Zmienna globalna zawiera : 5  
Watek -2 : Zmienna globalna zawiera : 6  
Watek 4 : Zmienna globalna zawiera : 6  
Watek 2 : Zmienna globalna zawiera : 6  
Watek 2 : Zmienna globalna zawiera : 6  
Watek 1 : Zmienna globalna zawiera : 6  
Watek 1 : Zmienna globalna zawiera : 6  
Watek 3 : Zmienna globalna zawiera : 6  
Watek 3 : Zmienna globalna zawiera : 6  
Watek 0 : Zmienna globalna zawiera : 6  
Watek 0 : Zmienna globalna zawiera : 6
```

Wątek master

Nie tracimy
pracy wykonanej
przez wątek
Master

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

omp_lock_t my_lock;

int main() {
    omp_init_lock(&my_lock);

#pragma omp parallel num_threads(4)  {
    int tid = omp_get_thread_num();
    int i, j;

    for (i = 0; i < 5; ++i) {
        omp_set_lock(&my_lock);
        printf_s("Thread %d - starting locked region\n", tid);
        printf_s("Thread %d - ending locked region\n", tid);
        omp_unset_lock(&my_lock);
    }
}

omp_destroy_lock(&my_lock);
}
```

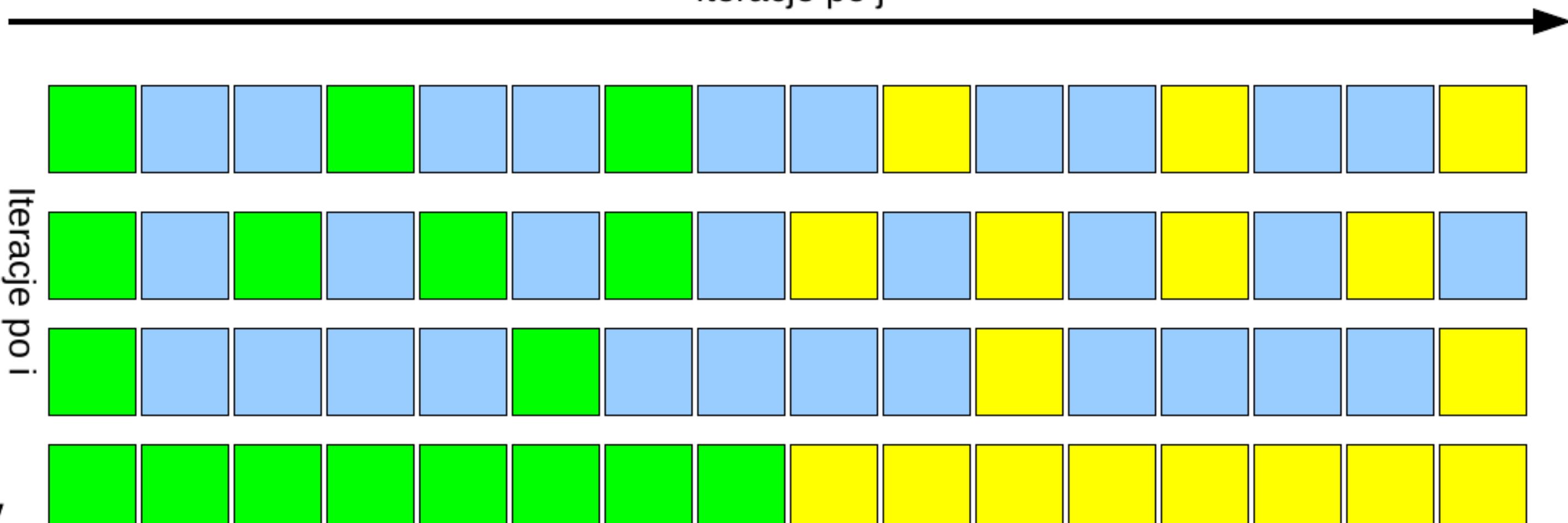
Lokalność danych (data locality) + zrównoleglenie

```
for ( int i = 0; i < delta_size; i++ )  
    zrównoleglemy wykonanie pętli po j  
        for ( int j = 0; j < N; j += delta[ i ] )  
            tablica[ j ] += 1.0;
```

N – bardzo duże!

Czyli: $N * \text{sizeof(int)} \gg \text{rozmiar pamięci cache CPU}$

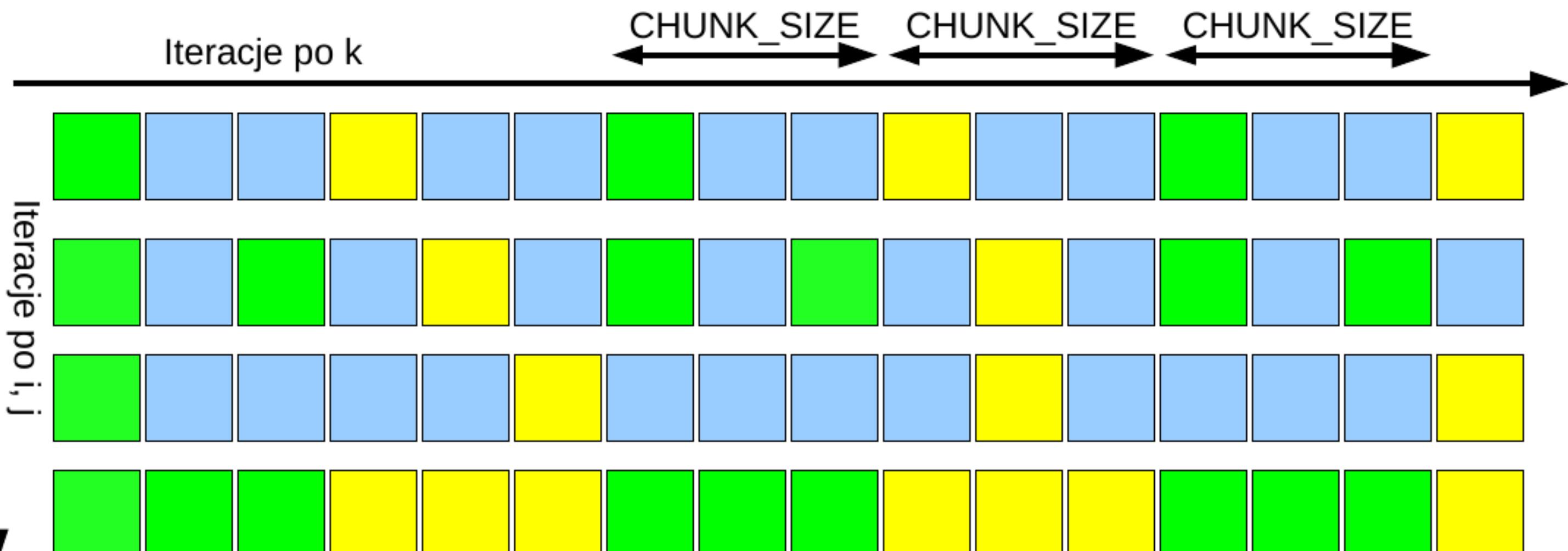
Iteracje po j



zrównoleglamy wykonanie pętli po k

```
for ( int k = 0; k < N; k += CHUNK_SIZE )  
    for ( int i = 0; i < delta_size; i++ ) {  
        st = // wyznaczenie indeksu początkowego  
        en = // wyznaczenie indeksu koncowego  
        for ( int j = st; j < en; j += delta[ i ] )  
            tablica[ j ] += 1.0;  
    }
```

num_threads * **CHUNK_SIZE** * sizeof(int) < rozmiar pamięci cache CPU



KONIEC