

Wykład Optymalizacja kodu

Między innymi na podstawie:

Intel® Guide for Developing Multithreaded Applications

January 16, 2012

Cele

- zmniejszenie udziału kodu sekwencyjnego
- poprawa granulacji
- równowaga obciążenia
- lokalność danych
- ograniczenie synchronizacji

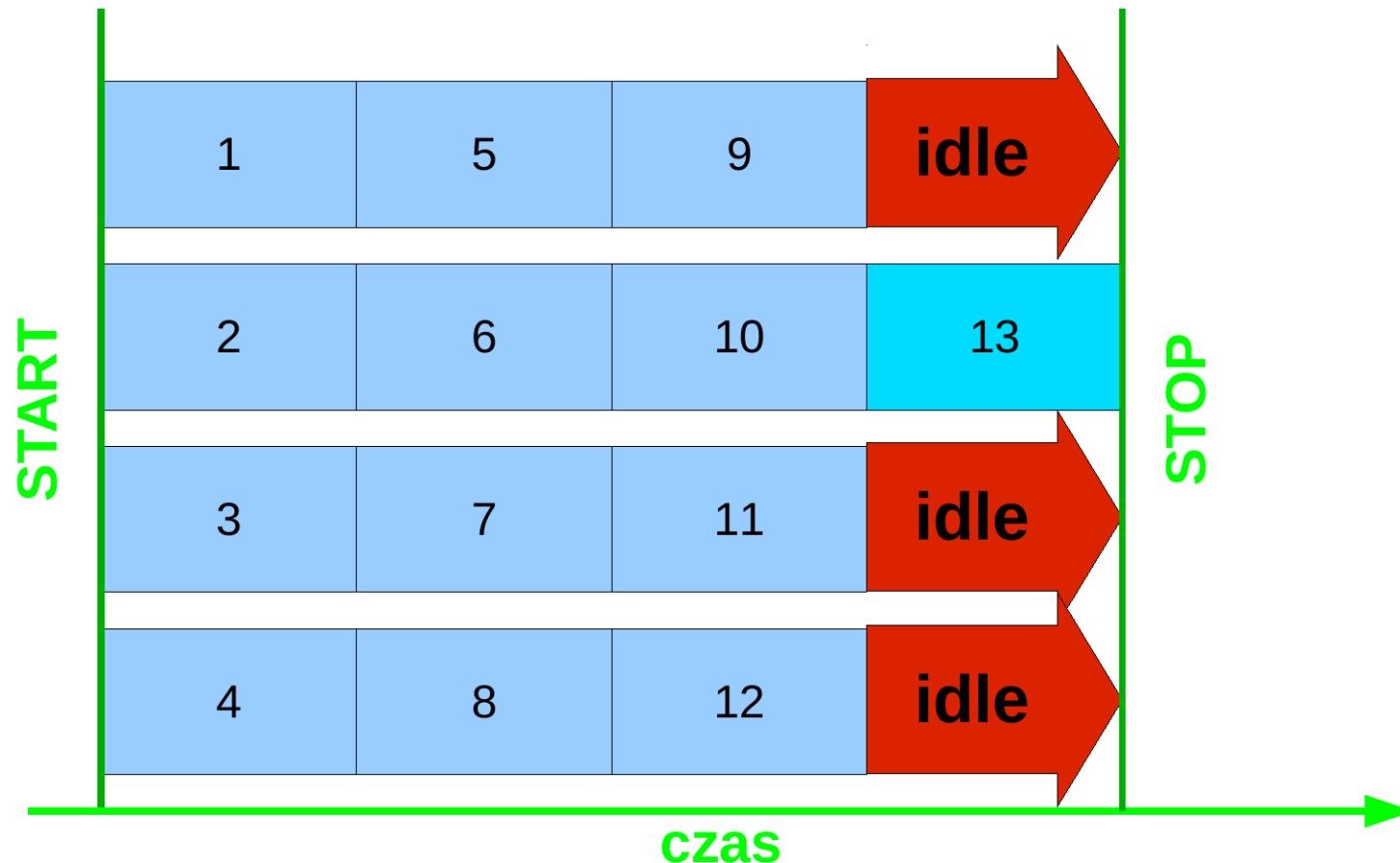
Jeśli program ma działać efektywnie to musimy zapanować nad narzutami.

Ogólna zasada: pętle zawierające dużą liczbę iteracji są dobrymi kandydatami do zrównoleglenia.

Zazwyczaj milcząco zakładamy, że w każdej iteracji pętli wykonywana jest zbliżona ilość pracy (czas wykonania jest praktycznie identyczny) – nie zawsze jest to jednak prawdą.

Przykład kodu, w którym trudno osiągnąć dobrą efektywność:

```
#pragma omp for
for ( i = 1; i <= 13; i++ )
    tu_coś_do_zrobienia();
```



Przykład:

```
for ( i = 1; i <= 5; i++ )  
    for ( j = 1; j <= 7; j++ )  
#pragma omp parallel for  
    for ( k = 1; k <= 500; k++ )  
        for ( l = 1; l <= 1000; l++ )  
            tu_coś_do_zrobienia();
```

Do zrównoleglenia wybieramy najbardziej zewnętrzna pętle o rozsądnie dużej liczbie powtórzeń.

Mögliwe jest także użycie klauzuli *collapse*.

Przykład:

```
#pragma omp for nowait
{
    for ( k = 1; k <= 500; k++ )
        for ( l = 1; l <= 1000; l++ )
            tu_coś_zrobienia();
}
```

Unikamy barier – oczywiście jeśli kolejne operacje są niezależne od wyniku tych pętli

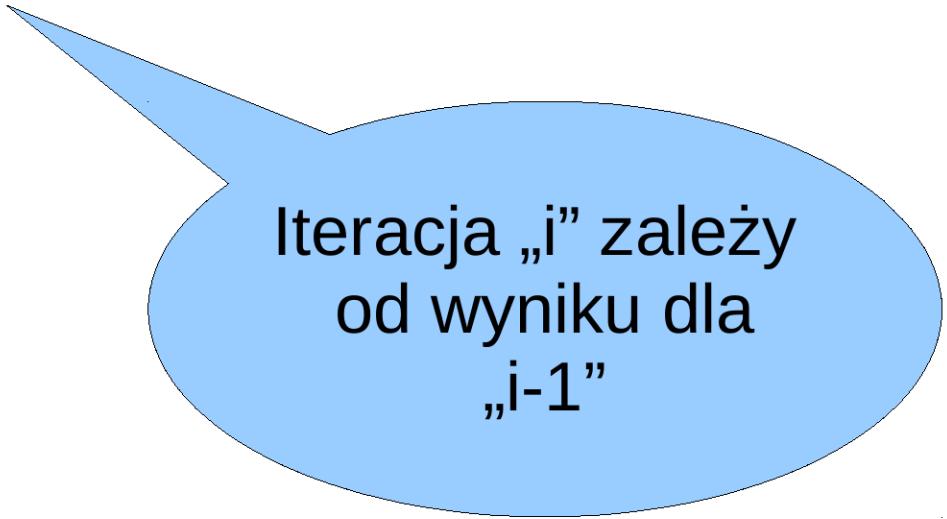
Problem tworzenia efektywnych programów równoległych znany jest od dawna i od dawna znane są recepty rozwiązuające typowe problemy z naszym kodem.

Dalej przedstawiono techniki

- dzielenia pętli w celu umożliwienia wykonania kodu równolegle
- dzielenia pętli w celu poprawy efektywnego użycia pamięci cache (lokalność danych)
- łącznia pętli

Dzielenie pętli (loop fission)

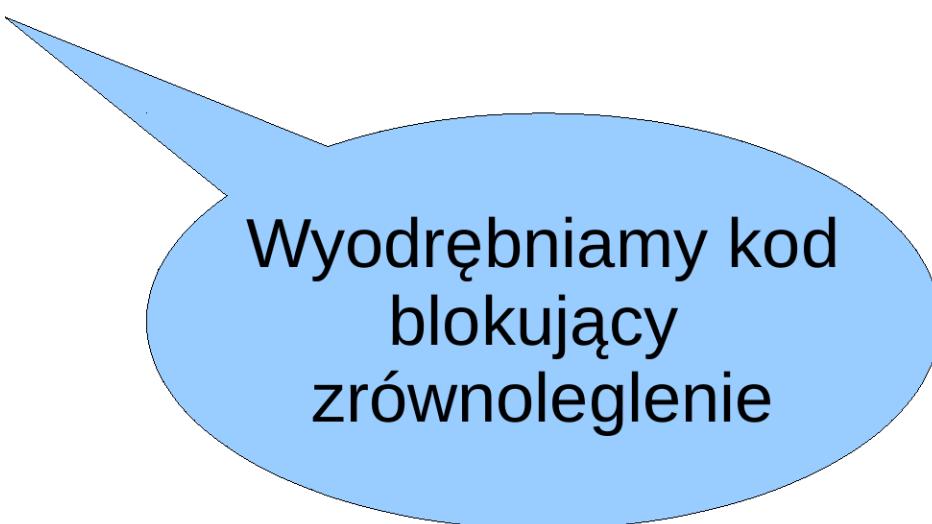
```
for ( int i = 1; i < N; i++ ) {  
    if ( b[ i ] > 0.0 )  
        a[ i ] = 2.0 * b[ i ];  
    else  
        a[ i ] = 2.0 * fabs( b[ i ] );  
    b[ i ] = a[ i - 1 ];  
}
```



Iteracja „i” zależy
od wyniku dla
„i-1”

Dzielenie pętli (loop fission)

```
for ( int i = 1; i < N; i++ ) {  
    if ( b[ i ] > 0.0 )  
        a[ i ] = 2.0 * b[ i ];  
    else  
        a[ i ] = 2.0 * fabs( b[ i ] );  
}  
  
for ( int i = 1; i < N; i++ ) {  
    b[ i ] = a[ i - 1 ];  
}
```



Wyodrębniamy kod blokujący zrównoleglenie

Dzielenie pętli (loop fission)

```
for ( int i = 1; i < N; i++ ) {  
    if ( b[ i ] > 0.0 )  
        a[ i ] = 2.0 * b[ i ];  
    else  
        a[ i ] = 2.0 * fabs( b[ i ] );  
}
```

```
for ( int i = 1; i < N; i++ ) {  
    b[ i ] = a[ i - 1 ];  
}
```

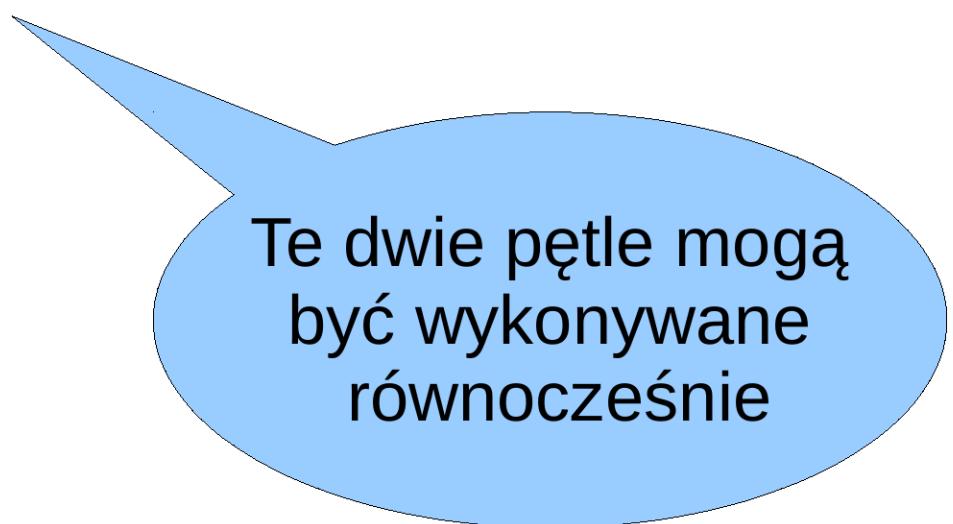
Dzielenie pętli (loop fission)

```
for ( int i = 1; i < N; i++ ) {  
    a[ i ] = a[ i - 1 ];  
    b[ i ] = b[ i - 1 ];  
}
```

Dzielenie pętli (loop fission)

```
for ( int i = 1; i < N; i++ ) {  
    a[ i ] = a[ i - 1 ];  
}
```

```
for ( int i = 1; i < N; i++ ) {  
    b[ i ] = b[ i - 1 ];  
}
```

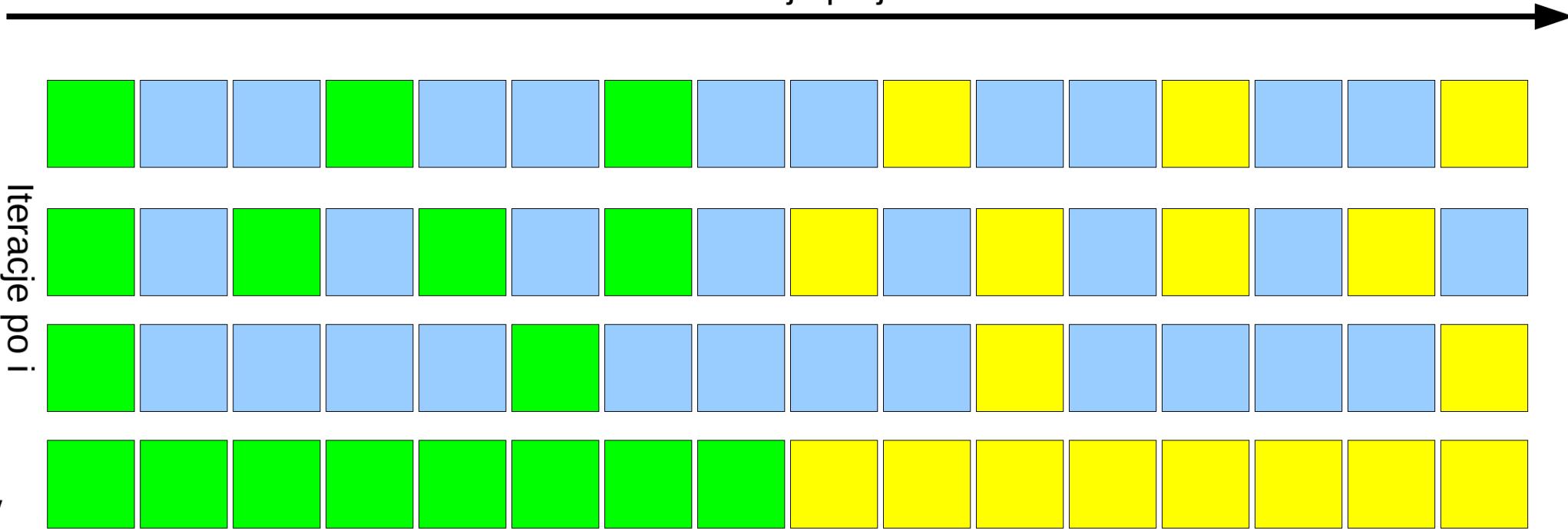


Te dwie pętle mogą być wykonywane równocześnie

Dzielenie pętli (loop fission) a lokalność danych (data locality)

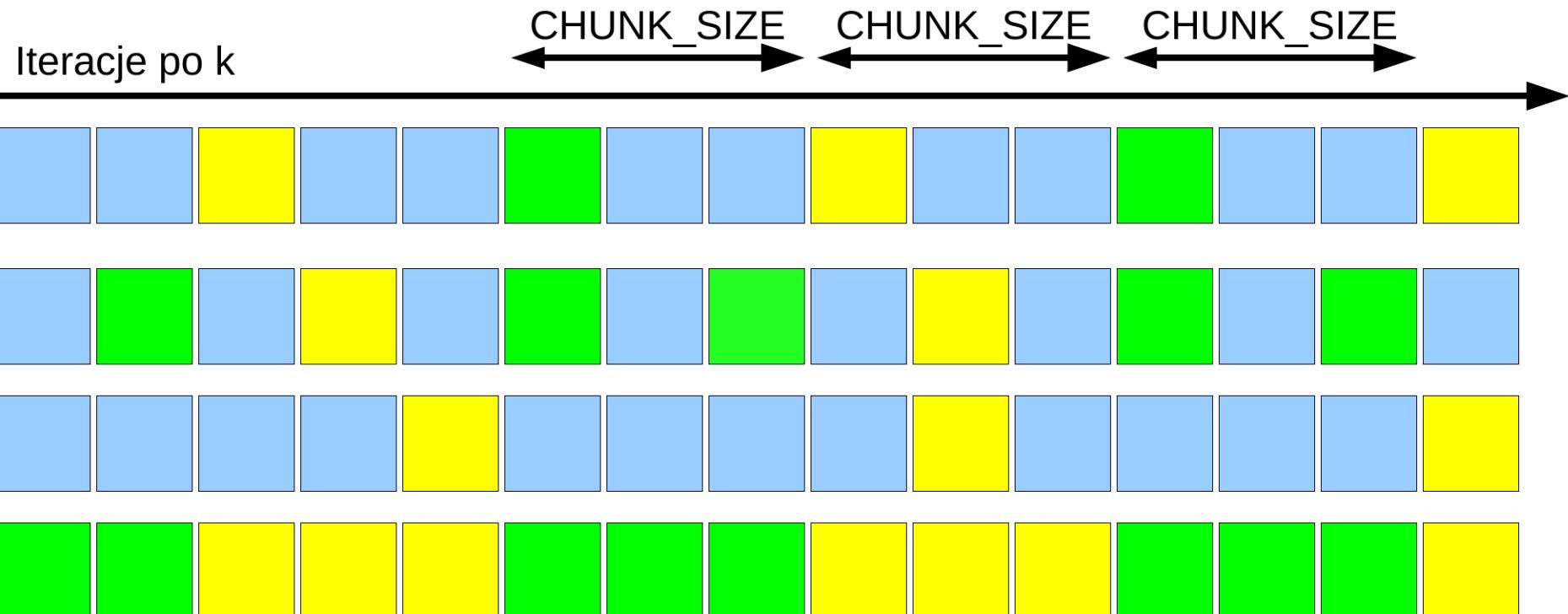
```
for ( int i = 0; i < delta_size; i++ )  
    zrównoleglamy wykonanie pętli po j  
        for ( int j = 0; j < N; j += delta[ i ] )  
            tablica[ j ] += 1.0;
```

Iteracje po j



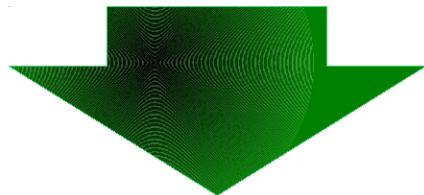
zrównoleglamy wykonanie pętli po k

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



Łączenie pętli

```
for ( k = 0; k < MALO; k ++ )  
    for ( i = 0; i < WZGLEDNIE_DUZO; i++ ) {  
        cos_to_zrobienia();  
    }
```



Można użyć
klauzuli
collapse

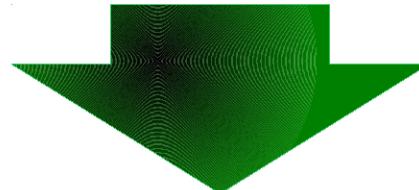
#pragma omp parallel for

```
#pragma omp parallel for  
for ( ki = 0; ki < MALO * WZGLEDNIE_DUZO; ki++ ) {  
    i = ki % WZGLEDNIE_DUZO;  
    k = ki / WZGLEDNIE_DUZO;  
    cos_to_zrobienia();  
}
```

Łączenie pętli w celu zwiększenia granulacji

```
#pragma omp parallel for
for ( int k = 0; k < N; k++ )
    cos_to_zrobienia_AAA();
```

```
#pragma omp parallel for
for ( int k = 0; k < N; k++ )
    cos_to_zrobienia_BBB();
```



```
#pragma omp parallel for
for ( int k = 0; k < N; k++ ) {
    cos_to_zrobienia_AAA();
    cos_to_zrobienia_BBB();
}
```

Zalecane jest używanie wersji równoległej tylko gdy zysk przewyższa narzuty. Decyzje można podjąć w trakcie wykonywania programu.

```
#pragma omp parallel for if ( N > limit )
for ( int k = 0; k < N; k++ )
    cos_to_zrobienia();
```

Granulacja – ilość rzeczywistej pracy w pojedynczym zadaniu równoległym. Ilość pracy pomiędzy aktami synchronizacji / komunikacji.

Trudno uniknąć problemów:

- drobnoziarnistość – narzuty na komunikacje
- gruboziarnistość – niezbalansowanie obciążenia

Najlepiej gdy program jest gruboziarnisty, a jednocześnie udaje się uniknąć niezbalansowania obciążenia

```
#pragma omp parallel
#pragma omp for schedule( static, CHUNK_SIZE )
```

<i>Chunk_Size</i>	<i>Time[s]</i>
1	0.799883
2	0.775192
4	0.742215
8	0.738204
[...]	
262144	0.736073
524288	0.768748
1048576	0.795836
2097152	0.890781
4194304	0.917303
8388608	1.38518

```
N = 100000000;
#pragma omp parallel for schedule( static, chunk )
for ( int i = 0; i < N; i++ )
    tablica[ i ] = exp( 20.0 * (double)i / N );
```

Rady

- **trzeba znać swoją aplikację** - ile pracy jest do wykonania w poszczególnych jej częściach oraz jakie są potrzeby komunikacyjne aplikacji (wymiana danych oraz synchronizacja)
- **trzeba znać platformę** - ile czasu zajmuje uruchomienie wielowątkowości / synchronizacja
- **trzeba znać narzędzia**

Dla dobrej efektywności ważne jest zrównoważenie obciążenia czyli minimalizacja czasu, w którym wątki nic nie robią

Sytuacja idealna – wszystkie wątki mają dokładnie tyle samo do zrobienia

Dlaczego wątek nic nie robi?

- bo nie ma co robić
- wykonuje operacje I/O
- czeka na dane, dostęp do współdzielonej zmiennej

Zrównoleglenie pętli for.

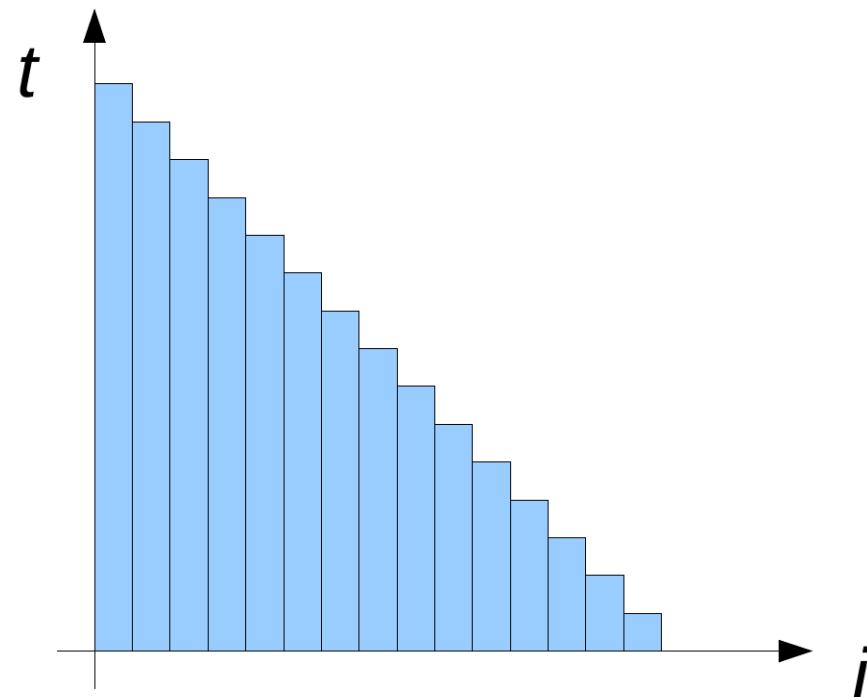
Jeśli czas wykonania poszczególnych iteracji jest porównywalny, to najprościej jest statycznie przedzielić pracę do wątków. Przydział statyczny jest domyślny. Każdy z wątków otrzymuje $1 / \text{liczba_wątków}$ części pracy

Jeśli iteracje się różnią, lepszy może być schemat dynamiczny. Wątek po wykonaniu powierzonej pracy (domyślnie jedna iteracja) „prosi” o nowy przydział.

Jest jeszcze schemat „guided” - na początku przydziały są większe, a na koniec coraz mniejsze. Unikamy w ten sposób niebalansowania obciążenia przy minimalizacji narzutu na komunikację.

Specjalny przypadek: ilość pracy **maleje** wraz z kolejnymi iteracjami. Schemat statyczny i guided dają złe wyniki.

```
#pragma omp parallel for schedule(static, 4)
for( int i = 0; i < size; i++ )
    for ( int j = size-1; j >= i; j-- )
        coś_do_zrobienia();
```



Nie ma jakiś magicznych mechanizmów na dynamiczny przydział zadań do wątków – to trzeba oprogramować.

Producer/Consumer

Producer – umieszcza zadania we współdzielonej strukturze typu kolejka.

Consumer – usuwa z kolejki zadanie i je wykonuje

Boss/Worker

Boss – osobny wątek, który rozdziela zadania

Worker – **kontaktuje się** z Boss-em w celu otrzymania zadania dla siebie. Ważne jest to, że w tym schemacie Boss **może** rozróżniać pracowników, którzy się z nim komunikują.

Samo zapewnienie pracy dla wielu wątków nie wystarcza – one muszą pracować efektywnie.

Optymalizacja kodu dla realizacji w wersji sekwencyjnej może wprowadzać zależności w danych, a to z kolei blokuje możliwość pracy równoległej – np. używanie wyników pośrednich w celu ograniczenia powtórzeń obliczeń.

Przykład: rozmycie obrazu poprzez uśrednienie pikseli w oknach 3x3

Pseudo-kod

```
for each pixel in (Image)
    sum = value of pixel
    for each neighbour of pixel
        sum += value of neighbour
    pixel = sum / 9
```

Pseudo-kod – bardzo wydajny, ale sekwencyjny

```
subroutine BlurLine( lineIn, lineOut )
    for each pixel j in lineIn
        lineOut[ j ] = ( pixel[ j - 1 ] +
                          pixel[ j ] + pixel[ j + 1 ] ) / 3

declare lineCache[ 3 ]
lineCache[ 0 ] = 0
BlurLine( line[ 1 ], lineCache[ 1 ] )
for each line i in Image
    BlurLine( line[i + 1], lineCache[i mod 3] )
    lineSum = lineCache[ 0 ]
                  + lineCache[ 1 ] + lineCache[ 2 ];
    line[ i ] = lineSum / 3
```

Operacje na wektorze

```
for ( i = 0; i < N; i++ ) {  
    obliczenia( &vector[ i ] );  
}
```

lub tak:

```
wsk = &vector[ 0 ];  
  
for ( i = 0; i < N; i++ ) {  
    obliczenia( wsk );  
    wsk++;  
}
```

Co zrobić?

Jeśli optymalizacji nie daje się utrzymać aby problem rozwiązać w całości, może da się ją zastosować po podziale problemu na mniejsze części

Przykład: istniejący algorytm rozmycia obrazu można zastosować po podziale obrazu na mniejsze fragmenty – każdy fragment liczymy niezależnie.

Co zrobić?

Zysk z optymalizacji może być mały (przykład ze wskaźnikiem), więc można z niej zrezygnować.

Kompilator może nasz kod zoptymalizować sam:

```
double sum = 0.0;
for ( int i = 0; i < N; i++ ) {
    tablica[ i ] = sin( exp( - tablica[ i ] ) );
    sum += tablica[ i ];
}
```

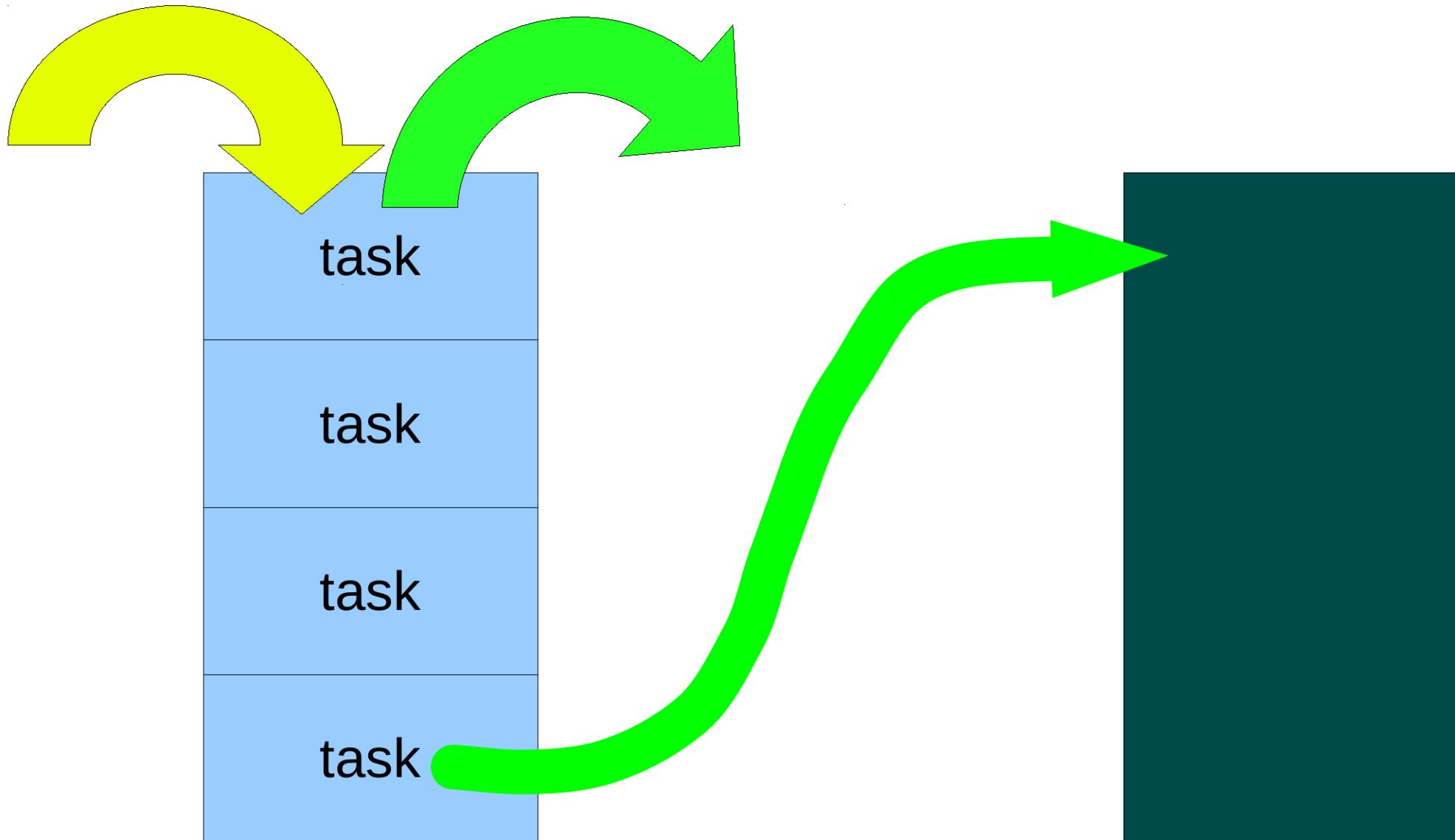
c++ source.cpp
ok. 9.55sek.

c++ -O3 source.cpp
ok. 9.16sek.

Użycie zadań (task) w miejsce wątków (thread)

- wątki – wie o nich OS. Muszą więc być dopasowane do sprzętu i to programista musi o to zadbać
- zadania – mamy pulę wątków gotowych do użycia w trakcie pracy aplikacji. Narzut mniejszy. OpenMP 3.0 (#pragma omp task)

Mechanizm podkradania pracy (work stealing) za pomocą kolejki dwustronnej. Efektywne użycie cache



```
#pragma omp parallel
{
    #pragma omp task
    {
        int n = omp_get_thread_num();
        cout << "FT ide spac      " << n << endl;
        sleep( 1 + n );
        cout << "FT wlasnie wstalem " << n << " -> " << omp_get_thread_num() << endl;
    }
}
```

```
#pragma omp task
{
    int n = omp_get_thread_num();

    cout << "ST " << n << " uruchamiam zadania zagniezdzzone..." << endl;
    for ( int i = 0; i < 3; i++ )
#pragma omp task
    {
        cout << "Zadanie " << i << " w zadaniu " << n << " -> " << omp_get_thread_num() << endl;
# pragma omp taskyield
        sleep(1);
        cout << "Zadanie " << i << " w zadaniu " << n << " -> " << omp_get_thread_num() << endl;
    }
}
```

```
#pragma omp taskwait

    cout << "Po taskwait " << n << " -> " << omp_get_thread_num() << endl;
} // task
} // parallel
```

```
#pragma omp parallel
```

```
{  
#pragma omp task  
{  
    int n = omp_get_thread_num();  
    cout << "FT ide spac " << n << endl;  
    sleep( 1 + n );  
    cout << "FT wlasnie wstalem " << n << " ->"  
}
```

```
#pragma omp task
```

```
{  
    int n = omp_get_thread_num();  
  
    cout << "ST " << n << " uruchamiam zadanie " << endl;  
    for ( int i = 0; i < 3; i++ )  
#pragma omp task  
{  
    cout << "Zadanie " << i << " w zadaniu " << endl;  
#pragma omp taskyield  
    sleep(1);  
    cout << "Zadanie " << i << " w zadaniu " << endl;  
}
```

```
#pragma omp taskwait
```

```
    cout << "Po taskwait " << n << " -> " << endl;  
} // task  
} // parallel
```

Dwa wątki



ST

Zadanie 1

Zadanie 2

Zadanie 3

ST

Zadanie 1

Zadanie 2

Zadanie 3

```
$ env OMP_NUM_THREADS=2 ./a.out
FT ide spac      0
ST 1 uruchamiam zadania zagniezdzone...
Zadanie 2 w zadaniu 1 -> 1
FT właśnie wstalem 0 -> 0
Zadanie 2 w zadaniu 1 -> 1
Zadanie 1 w zadaniu 1 -> 1
FT ide spac      0
Zadanie 1 w zadaniu 1 -> 1
FT właśnie wstalem 0 -> 0
Zadanie 0 w zadaniu 1 -> 1
ST 0 uruchamiam zadania zagniezdzone...
Zadanie 2 w zadaniu 0 -> 0
Zadanie 0 w zadaniu 1 -> 1
Zadanie 2 w zadaniu 0 -> 0
Po taskwait 1 -> 1
Zadanie 1 w zadaniu 0 -> 0
Zadanie 0 w zadaniu 0 -> 1
Zadanie 1 w zadaniu 0 -> 0
Zadanie 0 w zadaniu 0 -> 1
Po taskwait 0 -> 0
```

Każdy wątek generuje zadania.
Tu zlecane są one jawnie.

Niektóre zadania kończy inny wątek niż ten, który zaczął je wykonywać

```
$ env OMP_NUM_THREADS=2 ./a.out
FT ide spac      1
FT ide spac      0
FT właśnie wstalem 0 -> 0
ST 0 uruchamiam zadania zagnieżdzone...
Zadanie 2 w zadaniu 0 -> 0
FT właśnie wstalem 1 -> 1
ST 1 uruchamiam zadania zagnieżdzone...
Zadanie 2 w zadaniu 1 -> 1
Zadanie 2 w zadaniu 0 -> 0
Zadanie 1 w zadaniu 0 -> 0
Zadanie 2 w zadaniu 1 -> 1
Zadanie 1 w zadaniu 1 -> 1
Zadanie 1 w zadaniu 0 -> 0
Zadanie 0 w zadaniu 0 -> 0
Zadanie 1 w zadaniu 1 -> 1
Zadanie 0 w zadaniu 1 -> 1
Zadanie 0 w zadaniu 0 -> 0
Po taskwait 0 -> 0
Zadanie 0 w zadaniu 1 -> 1
Po taskwait 1 -> 1
```

Ten sam program
może wykonać
się inaczej.

taskwait
wymusza
zakończenie
zadań – dzieci
przed
kontynuacją pracy

Jak to działa

- gdy wątek napotyka konstrukcję **task** generowane jest jawnie zadanie
- zadanie może zostać przekazane do wykonania natychmiast lub odłożone na później – wszystko zależy od dostępności wątków
- wątki mogą zawieszać wykonywane zadania w punktach planowania/harmonogramowanie zadań (task scheduling point) i wykonywać inne zadania.
- zakończenie wszystkich jawnych zadań zleconych w danym bloku **parallel** jest gwarantowane przed wyjściem wątku master poza niejawną barierą na końcu bloku.
- zakończenie podzbioru jawnych zadań może zostać zlecone za pomocą **taskwait** lub **barrier** (task synchronization constructs)

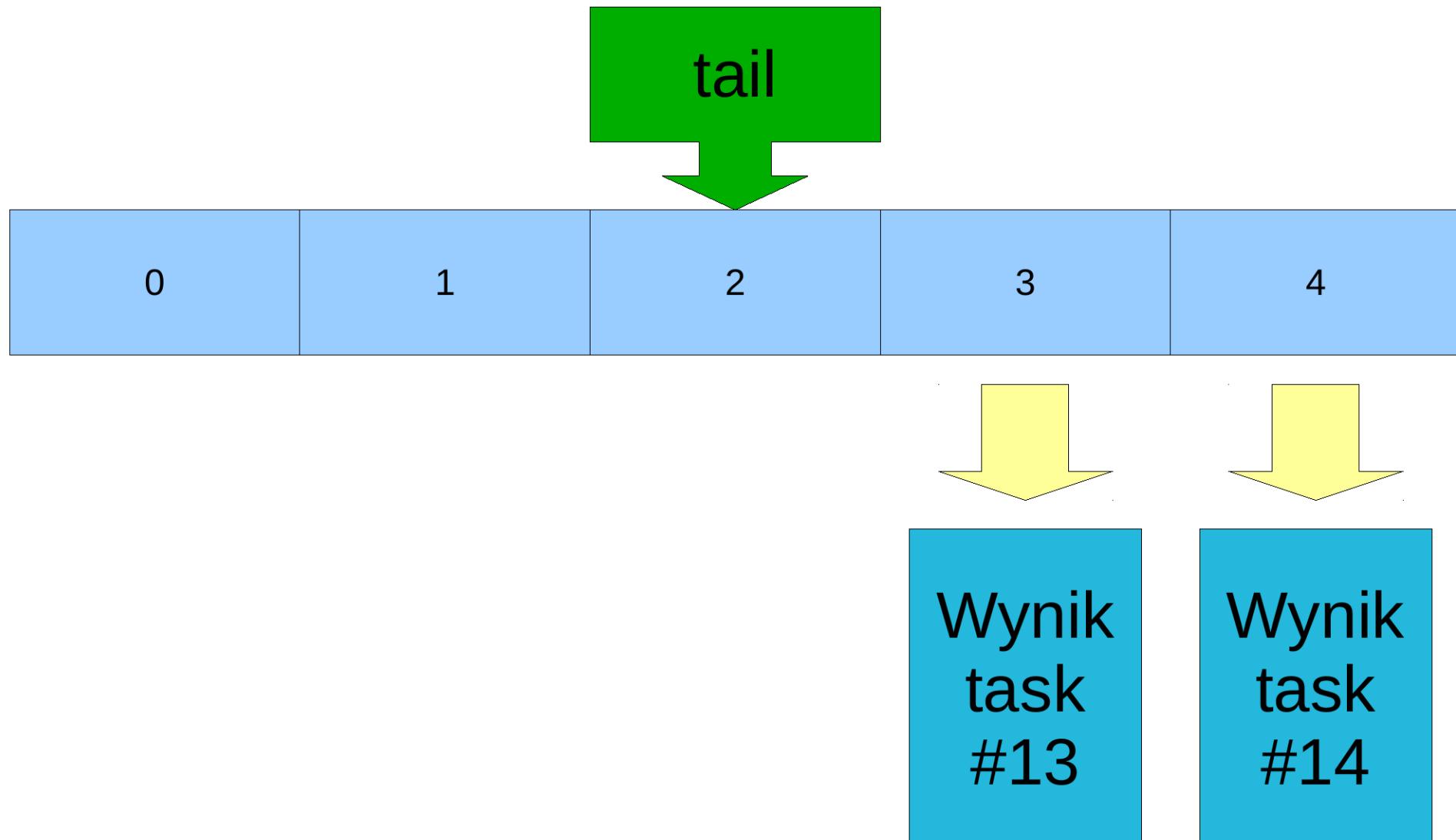
Przypadek danych uporządkowanych

- kodowanie dźwięku
- kodowanie obrazu
- kompresja bezstratna
- przetwarzanie strumienia danych pomiarowych

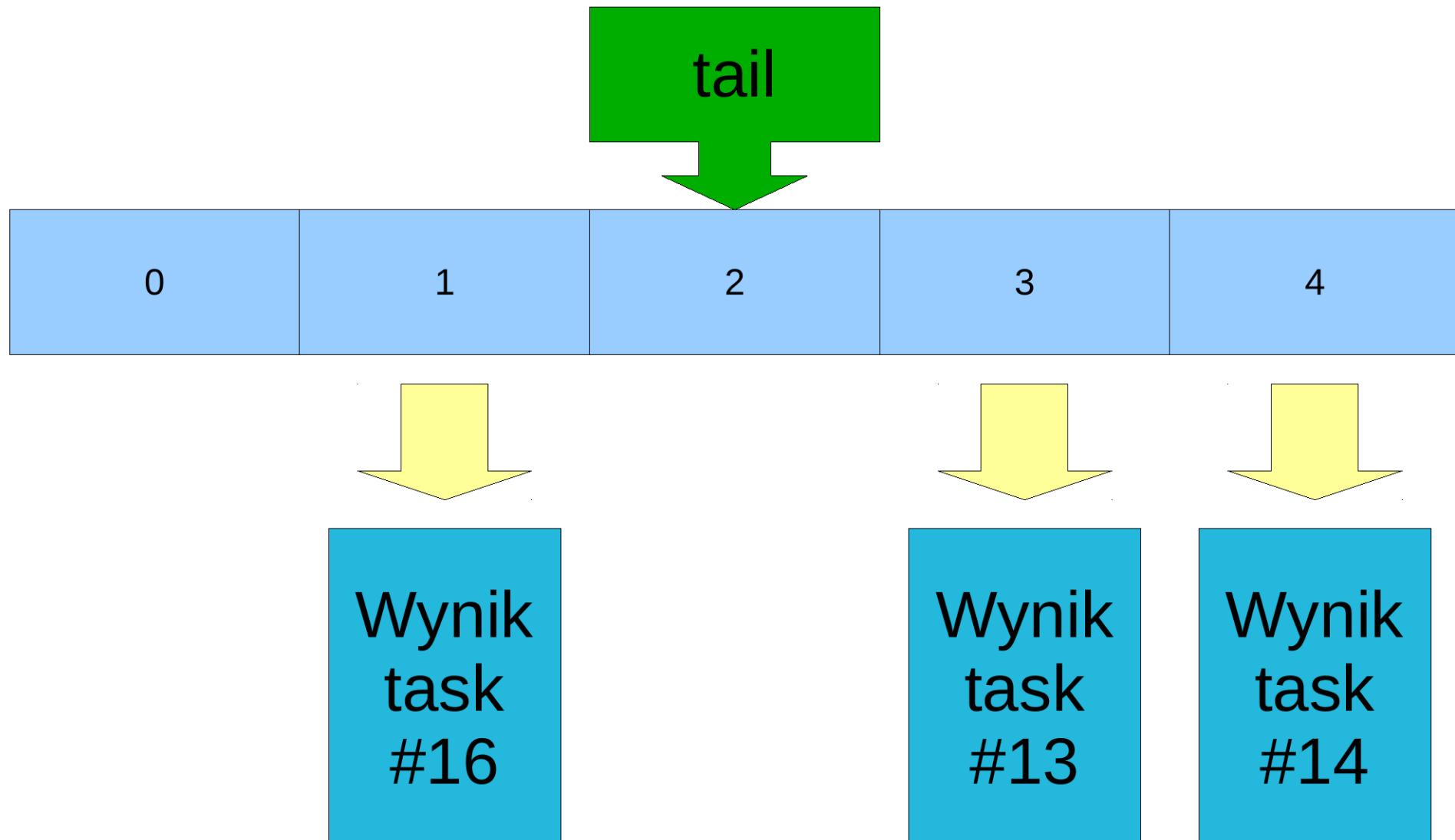
Wyzwania

- podział zadania na rozłączne podzbiory, które mogą być niezależnie i równocześnie przetwarzane
- upewnienie się, że dane nie są powielane i odczytywane we właściwej kolejności
- wyjście we właściwej kolejności bez względu na kolejność zakończenia przetwarzania danych
- to wszystko nie znając ilości danych

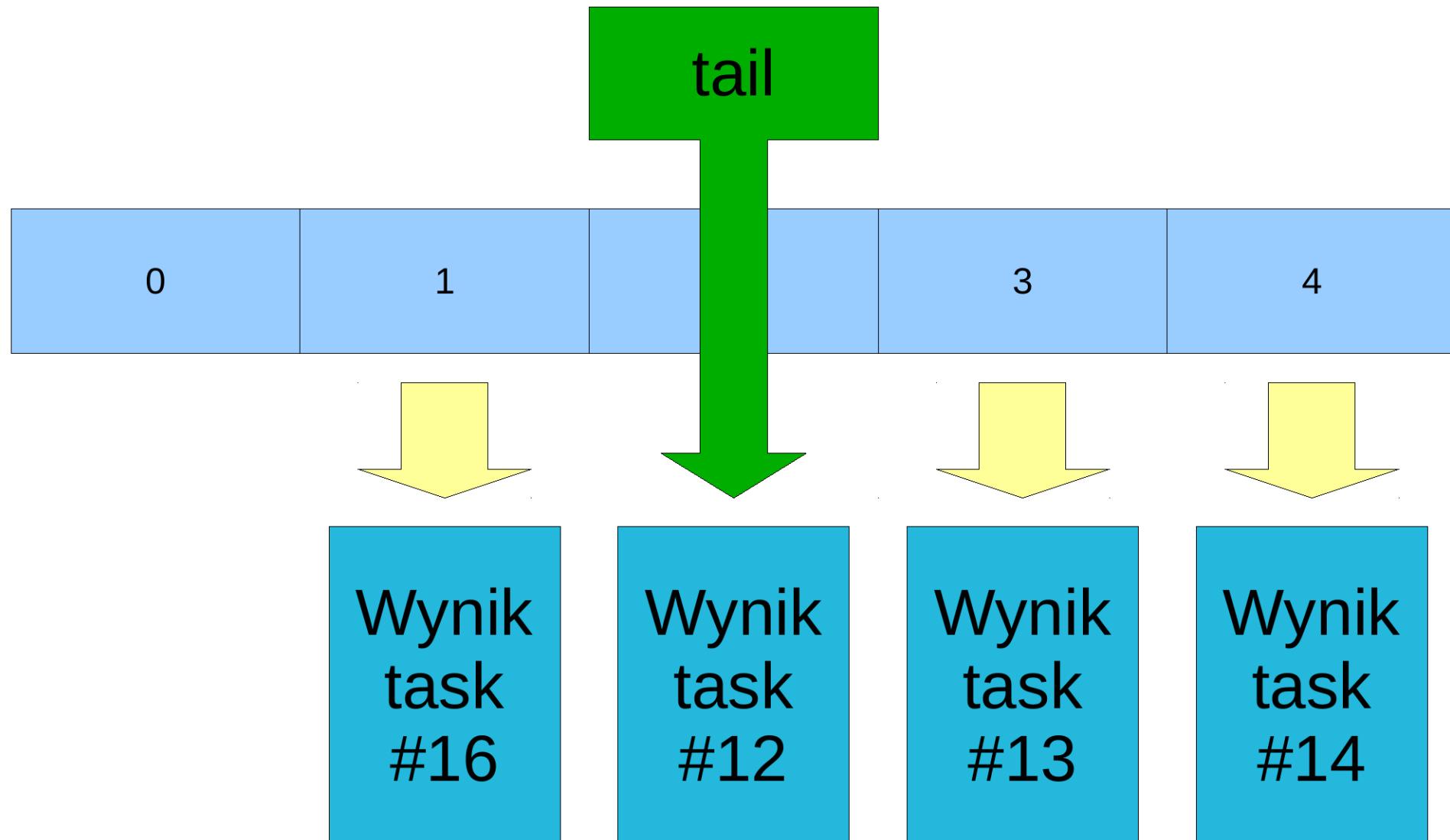
Przydział numerów sekwencyjnych do zadań + organizacja bufora wyjściowego



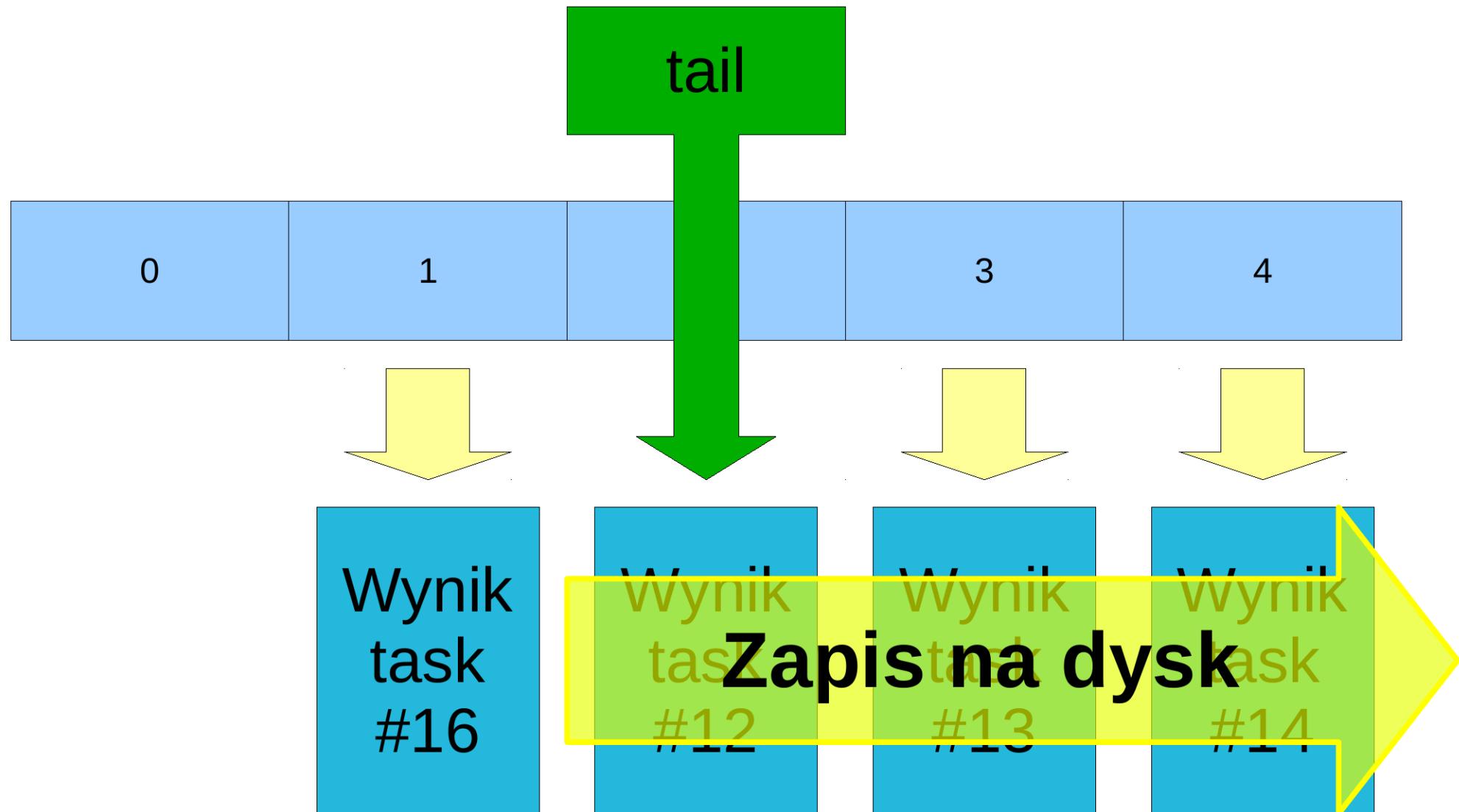
Przydział numerów sekwencyjnych do zadań + organizacja bufora wyjściowego



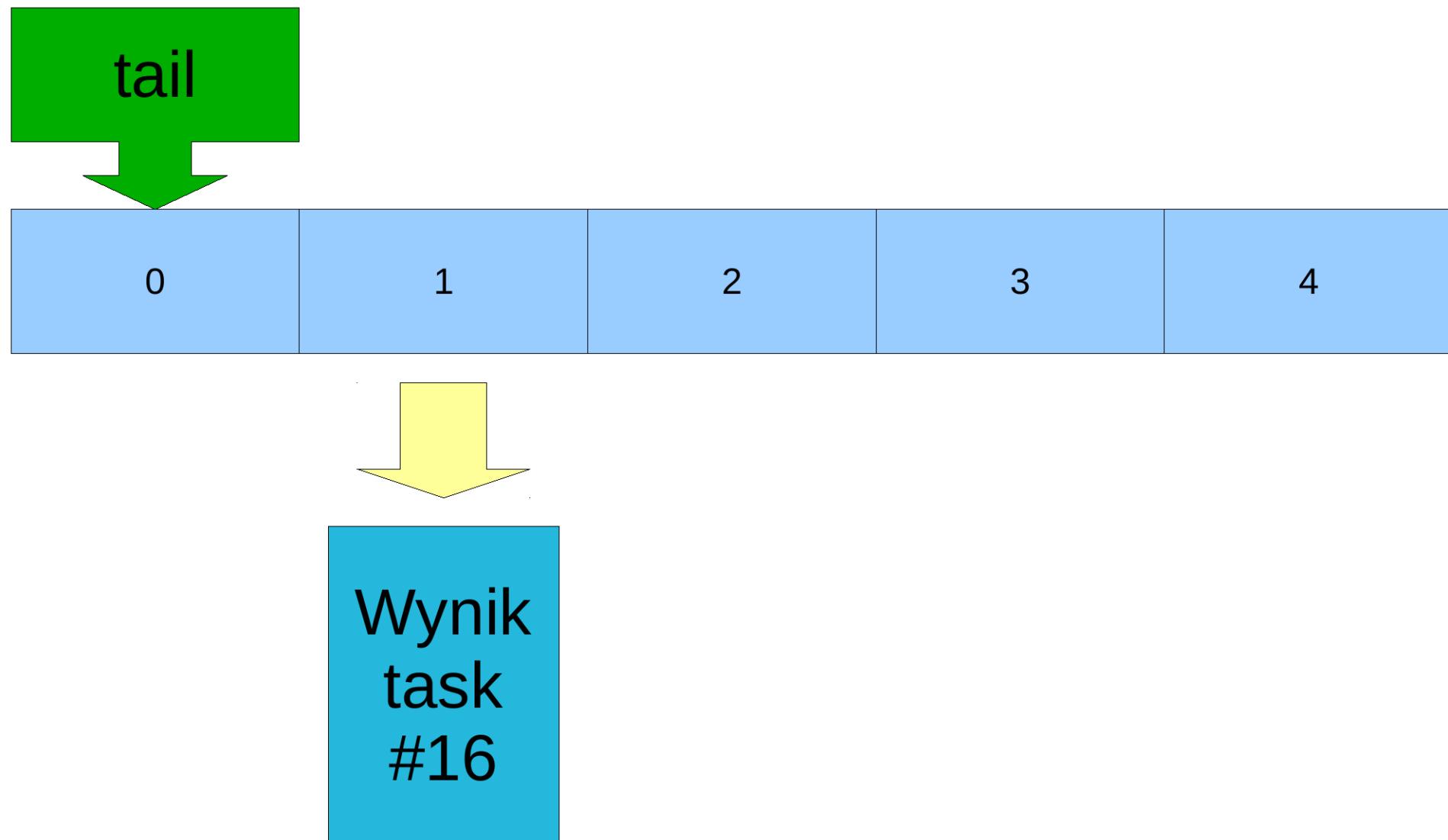
Przydział numerów sekwencyjnych do zadań + organizacja bufora wyjściowego



Przydział numerów sekwencyjnych do zadań + organizacja bufora wyjściowego



Przydział numerów sekwencyjnych do zadań + organizacja bufora wyjściowego



Przydział numerów sekwencyjnych do zadań + organizacja bufora wyjściowego

- do wyznaczenie indeksu wystarczy używać reszty z dzielenia
- współdzielony bufor musi być chroniony przed równoczesnym dostępem
- ilość slotów > ilość wątków
- wątek powinien posiadać prealokowane bufory na wyniki
- operować na wskaźnikach, a nie kopiować bloki pamięci
- jednoczesna próba odczytu prowadzi do bardzo nieefektywnego używania dysku

Sekcje krytyczne – obszar kodu, który może być realizowany tylko przez jeden wątek – typowy przypadek to operacje na współdzielonych zmiennych. Zapewniają integralność danych

Rozmiar sekcji krytycznej – czas spędzony na realizację jej kodu

Generalnie: Im sekcje krytyczne mniejsze tym lepiej, ale bez przesady – obsługa sekcji (blokady) także zajmuje czas

Oczekiwanie na dostęp:

spin-wait – wątek aktywnie sprawdza czy nie ma już dostępu do sekcji. Wątek mocno obciąża tym testem zasoby systemu

context switch – wątek czekający jest zamieniany przez OS na wątek aktywny – dostęp do zasobów jest przydzielany dla kogoś innego. To trwa...

Intel® Hyper-Threading Technology lub odpowiednik – tu trzeba uważać, bo część rdzeni jest wyłącznie tworem logicznym, więc spin-wait na pewno obniża wydajność

Przykład:

```
for( int j = 0; j < size; j++ ) {  
    suma1 += sin( 6.28 * j / size + i / rep );  
    suma2 += cos( 6.28 * j / size + i / rep );  
    suma3 += sin( 6.28 * j / size ) + cos( 6.28 * i / rep );  
}
```

Delta T = 14.56

S1 -18899.2

S2 -44900.1

S3 -43739.8

Przykład:

```
#pragma omp parallel for shared( suma1, suma2, suma3 )
for( int j = 0; j < size; j++ ) {
#pragma omp critical (s1)
    suma1 += sin( 6.28 * j / size + i / rep );
#pragma omp critical (s2)
    suma2 += cos( 6.28 * j / size + i / rep );
#pragma omp critical (s3)
    suma3 += sin( 6.28 * j / size ) + cos( 6.28* i / rep );
}
```

Delta T = **35.465 (było 14.6)**

S1 -18899.2
S2 -44900.1
S3 -43739.8

Przykład:

```
#pragma omp parallel for shared( suma1, suma2, suma3 )
for( int j = 0; j < size; j++ ) {
#pragma omp critical
{
    suma1 += sin( 6.28 * j / size + i / rep );
    suma2 += cos( 6.28 * j / size + i / rep );
    suma3 += sin( 6.28 * j / size ) + cos( 6.28* i / rep );
}
}
```

Delta T = **60.21 (było 14.6)**

S1 -18899.2
S2 -44900.1
S3 -43739.8

Przykład:

```
#pragma omp parallel for shared( suma1, suma2, suma3 )
                           private( s1, s2, s3)
for( int j = 0; j < size; j++ ) {
    s1 += sin( 6.28 * j / size + i / rep );
    s2 += cos( 6.28 * j / size + i / rep );
    s3 += sin( 6.28 * j / size ) + cos( 6.28* i / rep );
#pragma omp critical
{
    suma1 += s1;
    suma2 += s2;
    suma3 += s3;
}
}
```

Delta T = **16.33 (było 14.6)**

S1 -18899.2
S2 -44900.1
S3 -43739.8

Przykład:

```
#pragma omp parallel for reduction(+ : suma1, suma2, suma3 )
for( int j = 0; j < size; j++ ) {
    suma1 += sin( 6.28 * j / size + i / rep );
    suma2 += cos( 6.28 * j / size + i / rep );
    suma3 += sin( 6.28 * j / size ) + cos( 6.28* i / rep );
}
```

Delta T = **7.399** (było 14.6)

S1 -18899.2

S2 -44900.1

S3 -43739.8

Generalnie: nie pisać własnego kodu do synchronizacji wątków

Używać gotowych rozwiązań z wybranego API biblioteki obsługującej wątki

Jeśli można to używać blokad, które nie zablokują wykonywania wątku, lecz zakończą się komunikatem o niemożności uzyskania dostępu do współdzielonego zasobu.

Tzn. niejako **sprawdzamy** czy natychmiast można zyskać dostęp do zasobu – jeśli tak, to wykonuje się kod w sekcji krytycznej; jeśli nie, to wątek nie jest zatrzymywany, tylko może wykonywać jakąś inną użyteczną pracę spoza sekcji krytycznej.

Stos systemowy jest współdzielony – operacje typu malloc() są kosztowne, bo wymagają współudziału OS.

Można używać innych funkcji... np. z OpenMP + Intel® Compiler czy Win32* API. Dają one możliwość przydziału stosu do wątku i uniknięcia globalnej synchronizacji

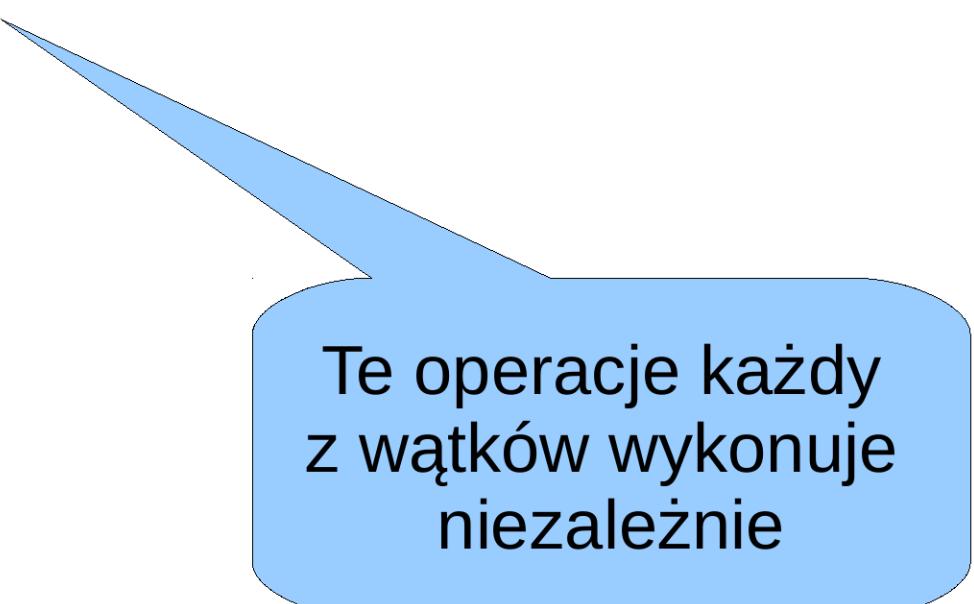
Użycie zmiennych lokalnych wątków w celu ograniczenia synchronizacji:

```
#pragma omp parallel for shared( suma )
for( int j = 0; j < size; j++ ) {
#pragma omp atomic
    suma += j;
}
```

Delta T = 3.71617

Użycie zmiennych lokalnych wątków w celu ograniczenia synchronizacji:

```
#pragma omp parallel shared( suma )  
    firstprivate( suma1 )  
{  
#pragma omp for  
    for( int j = 0; j < size; j++ ) {  
        suma1 += j;  
    }  
#pragma omp atomic  
    suma += suma1;  
}  
  
Delta T = 0.034968
```



Te operacje każdy z wątków wykonuje niezależnie

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int size_x = 300;
int size_y = 100000;
int main( void )
{
    double **array = new double* [ size_x ];
    for ( int i = 0; i < size_x; i++ ) {
        array[ i ] = new double [ size_y ];
        for ( int j = 0; j < size_y; j++ )
            array[ i ][ j ] = i + j / 10.0;
    }
    double sum = 0.0;
    for ( int r = 0; r < 10; r++ )
        for ( int i = 0; i < size_x; i++ )
            for ( int j = 0; j < size_y; j++ )
                sum += array[ i ][ j ];
    cout << "sum = " << sum << endl;
}
```

To kolejny przykład pokazujący jak ważne jest efektywne użycie pamięci cache

```
for ( int i = 0; i < size_x; i++ )  
    for ( int j = 0; j < size_y; j++ )  
        sum += array[ i ][ j ];
```

```
$ time ./a.out  
sum = 1.54484e+12
```

```
real 0m4.542s  
user 0m3.737s  
sys 0m0.738s
```

```
for ( int i = 0; i < size_y; i++ )  
    for ( int j = 0; j < size_x; j++ )  
        sum += array[ j ][ i ];
```

```
$ time ./a.out  
sum = 1.54484e+12
```

```
real 0m24.405s  
user 0m23.439s  
sys 0m0.861s
```

4.5 sekundy

24.4 sekundy

```
int size = 50000000;  
  
double *a = new double [ size ];  
double *b = new double [ size ];  
double *delta = new double[ size ];  
for ( int j = 0; j < size; j++ ) {  
    a[ j ] = 40.0 * random( ) / RAND_MAX;  
    b[ j ] = 40.0 * random( ) / RAND_MAX;  
    delta[ j ] = 0.0;  
}  
  
struct timeval ti; gettimeofday( &ti, NULL );  
  
for ( int j = 0; j < size; j++ ) { // petla 1  
    delta[ j ] = fabs( a[ j ] - b[ j ] );  
    if ( delta[ j ] > 20.0 ) delta[ j ] = 2.06e-9;  
    else delta[ j ] = exp( - delta[ j ] );  
}  
show( &ti );  
  
for ( int j = 0; j < size; j++ ) { // petla 2  
    delta[ j ] = fabs( a[ j ] - b[ j ] );  
    delta[ j ] = exp( - delta[ j ] );  
}  
show( &ti );  
  
for( int j = 0; j < size; j++ ) { // petla 3  
    delta[ j ] = exp( -fabs( a[ j ] - b[ j ] ) );  
}  
show( &ti );
```

Pętle 1, 2 i 3 są równoważne
Ale nie działają równie szybko...

```
> c++ -O3 code60.vect.cpp
```

```
> ./a.out
```

```
Delta T = 3.0679          // petla 1    1 miejsce  
Delta T = 6.80106 ( 3.73316 ) // petla 2    3 miejsce  
Delta T = 10.5235 ( 3.72245 ) // petla 3    2 miejsce
```



```
> icc -fast code60.vect.cpp          // intel C compiler
```

```
code60.vect.cpp(43): (col. 4) remark: PARTIAL LOOP WAS VECTORIZED.  
code60.vect.cpp(43): (col. 4) remark: PARTIAL LOOP WAS VECTORIZED.  
code60.vect.cpp(54): (col. 4) remark: PARTIAL LOOP WAS VECTORIZED.  
code60.vect.cpp(54): (col. 4) remark: PARTIAL LOOP WAS VECTORIZED.  
code60.vect.cpp(63): (col. 4) remark: LOOP WAS VECTORIZED.
```

```
> ./a.out
```

```
Delta T = 0.753245          // petla 1    3 miejsce  
Delta T = 1.32093 ( 0.567669 ) // petla 2    2 miejsce  
Delta T = 1.75803 ( 0.437101 ) // petla 3    1 miejsce
```



```
void funkcja( int id ) {  
#pragma omp for  
    for ( int i = 0; i < 3; i++ ) {  
        cout << "To ja watek nr " << id << " i = " << i << endl;  
        usleep( 1000000 );  
    }  
  
#pragma omp for  
    for ( int i = 0; i < 50; i++ ) {  
        usleep( i * 2000 );  
        cout << "To ja watek nr " << id << " i = " << i << endl;  
    }  
  
}  
  
int main ( void ) {  
#pragma omp parallel  
{  
    for ( int i = 0; i < 5; i++ )  
        funkcja( omp_get_thread_num() );  
}  
  
long int sum = 0;  
#pragma omp parallel for schedule( static, 1 )  
    for ( int i = 0; i < 1000000; i++ ) {  
#pragma omp atomic  
    sum += i;  
}  
}
```

Trzy iteracje
a liczba
wątków
parzysta

Ilość pracy
rośnie wraz
z „i” a
podział pracy
domyślny

Duża liczba iteracji,
podział pracy
statyczny po jednej
iteracji. We wnętrzu
pętli operacja
atomowa.

Kompilacja kodu

```
$ kinst-ompp c++ -fopenmp c2prof.cpp
```

Uruchomienie

```
$ env OMP_NUM_THREADS=2 ./a.out  
$ env OMP_NUM_THREADS=4 ./a.out
```

Raport

```
$ cat env.2-0.ompp.txt  
$ cat env.4-0.ompp.txt
```

```
To ja watek nr 0 i = 0
To ja watek nr 1 i = 2
To ja watek nr 0 i = 1

To ja watek nr 0 i = 0
To ja watek nr 0 i = 1
To ja watek nr 0 i = 2
To ja watek nr 1 i = 25
To ja watek nr 0 i = 7
To ja watek nr 0 i = 8
To ja watek nr 0 i = 9
To ja watek nr 1 i = 26
To ja watek nr 0 i = 12
To ja watek nr 0 i = 13
To ja watek nr 1 i = 28
To ja watek nr 1 i = 34
To ja watek nr 0 i = 24
To ja watek nr 1 i = 44
To ja watek nr 1 i = 45
To ja watek nr 1 i = 46
To ja watek nr 1 i = 47
To ja watek nr 1 i = 48
To ja watek nr 1 i = 49
```

ompP General Information

Start Date : Mon Apr 09 11:39:28 2012
End Date : Mon Apr 09 11:39:52 2012
Duration : 24.74 sec
Application Name: env
Type of Report : final
User Time : 0.67 sec
System Time : 4.48 sec
Max Threads : 2
ompP Version : 0.7.0
ompP Build Date : Apr 6 2012 20:51:48
PAPI Support : not available

ompP Region Overview

PARALLEL: 1 region:

- * R00003 c2prof.cpp (22-26)

PARALLEL LOOP: 1 region:

- * R00004 c2prof.cpp (29-33)

LOOP: 2 regions:

- * R00001 c2prof.cpp (8-12)
- * R00002 c2prof.cpp (14-18)

ATOMIC: 1 region:

- * R00005 c2prof.cpp (31-32)

ompP Callgraph

Inclusive (%)	Exclusive (%)
24.74 (100.0%)	0.00 (0.00%)
19.27 (77.90%)	0.00 (0.00%)
10.00 (40.43%)	10.00 (40.43%)
9.27 (37.47%)	9.27 (37.47%)
5.47 (22.10%)	2.96 (11.98%)
2.50 (10.12%)	2.50 (10.12%)

PARALLEL
LOOP
LOOP
PARLOOP
ATOMIC

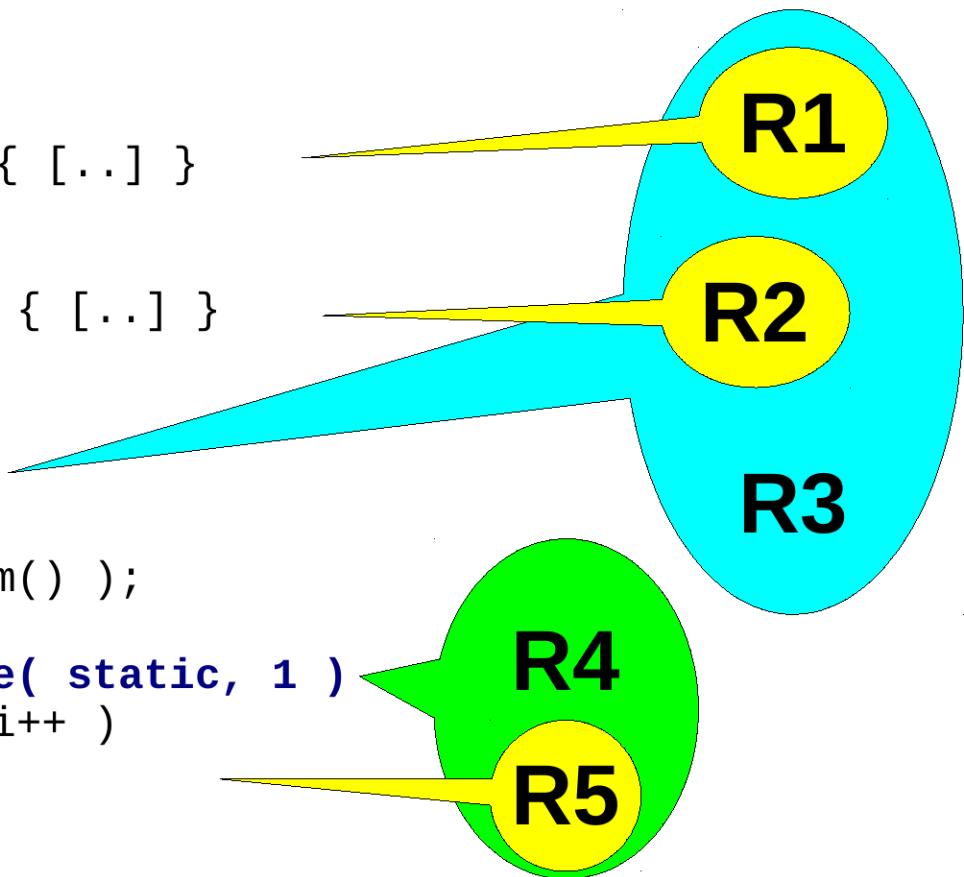
[env: 2 threads]
|-R00003 c2prof.cpp (22-26)
| | -R00001 c2prof.cpp (8-12)
| | +-R00002 c2prof.cpp (14-18)
+-R00004 c2prof.cpp (29-33)
+-R00005 c2prof.cpp (31-32)

```
void funkcja( int id ) {
#pragma omp for
for ( int i = 0; i < 3; i++ ) { [...] }

#pragma omp for
for ( int i = 0; i < 50; i++ ) { [...] }

int main ( void ) {
#pragma omp parallel
for ( int i = 0; i < 5; i++ )
funkcja( omp_get_thread_num() );

#pragma omp parallel for schedule( static, 1 )
for ( int i = 0; i < 1000000; i++ )
#pragma omp atomic
sum += i;
}
```



Profiler ompP - raport

Optym/68

ompP Flat Region Profile (inclusive data)

R00003 c2prof.cpp (22-26) PARALLEL

TID	execT	execC	bodyT	exitBarT	startupT	shutdwnT	taskT
0	19.27	1	19.27	0.00	0.00	0.00	0.00
1	19.27	1	19.27	0.00	0.00	0.00	0.00
SUM	38.54	2	38.54	0.00	0.00	0.00	0.00

R00001 c2prof.cpp (8-12) LOOP

TID	execT	execC	bodyT	exitBarT	taskT
0	10.00	5	10.00	0.00	0.00
1	10.00	5	5.00	5.00	0.00
SUM	20.00	10	15.00	5.00	0.00

R00002 c2prof.cpp (14-18) LOOP

TID	execT	execC	bodyT	exitBarT	taskT
0	9.27	5	3.03	6.24	0.00
1	9.27	5	9.27	0.00	0.00
SUM	18.54	10	12.29	6.24	0.00

R00004 c2prof.cpp (29-33) PARALLEL LOOP

TID	execT	execC	bodyT	exitBarT	startupT	shutdwnT	taskT
0	5.47	1	5.47	0.00	0.00	0.00	0.00
1	5.47	1	5.46	0.01	0.00	0.00	0.00
SUM	10.93	2	10.92	0.01	0.00	0.00	0.00

R00005 c2prof.cpp (31-32) ATOMIC

TID	execT	execC
0	2.70	500000
1	2.56	500000
SUM	5.26	1000000

 ompP Overhead Analysis Report

Total runtime (wallclock) : 24.74 sec [2 threads]
 Number of parallel regions : 2
 Parallel coverage : 24.74 sec (**100.00%**)

Parallel regions sorted by wallclock time:

	Type	Location	Wallclock (%)
R00003	PARALLEL	c2prof.cpp (22-26)	19.27 (77.90)
R00004	PARLOOP	c2prof.cpp (29-33)	5.47 (22.10)
		SUM	24.74 (100.00)

Overheads wrt. each individual parallel region:

	Total	Ovhds (%)	=	Synch (%)	+ Imbal (%)	+ Limpar (%)	+ Mgmt (%)
R00003	38.54	11.24 (29.18)		0.00 (0.00)	11.24 (29.18)	0.00 (0.00)	0.00 (0.00)
R00004	10.93	5.01 (45.87)		5.01 (45.80)	0.01 (0.06)	0.00 (0.00)	0.00 (0.01)

Overheads wrt. whole program:

	Total	Ovhds (%)	=	Synch (%)	+ Imbal (%)	+ Limpar (%)	+ Mgmt (%)
R00003	38.54	11.24 (22.73)		0.00 (0.00)	11.24 (22.73)	0.00 (0.00)	0.00 (0.00)
R00004	10.93	5.01 (10.14)		5.01 (10.12)	0.01 (0.01)	0.00 (0.00)	0.00 (0.00)
SUM	49.47	16.26 (32.86)		5.01 (10.12)	11.25 (22.74)	0.00 (0.00)	0.00 (0.00)

KONIEC