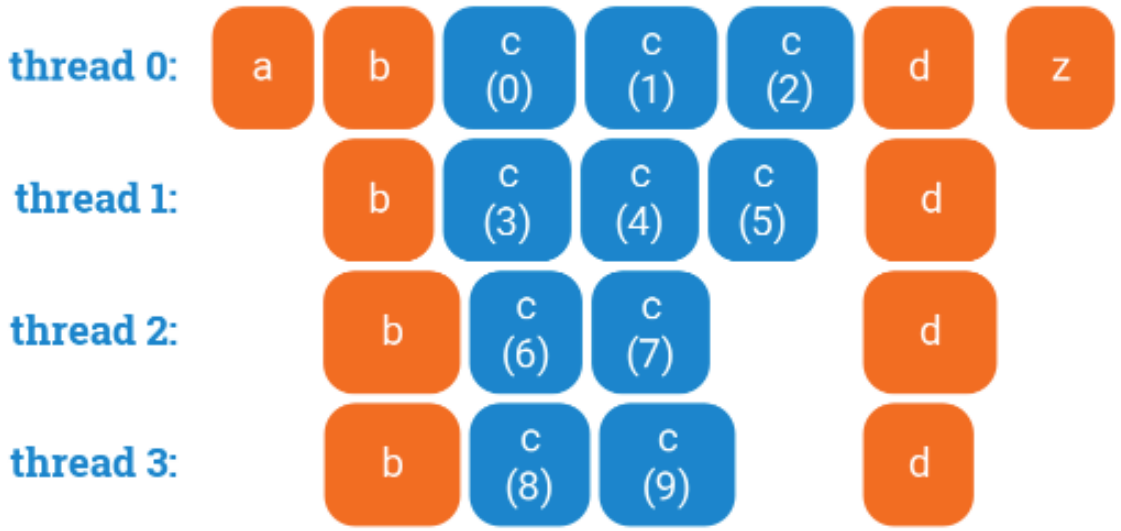
Pętla for – punkt synchronizacji

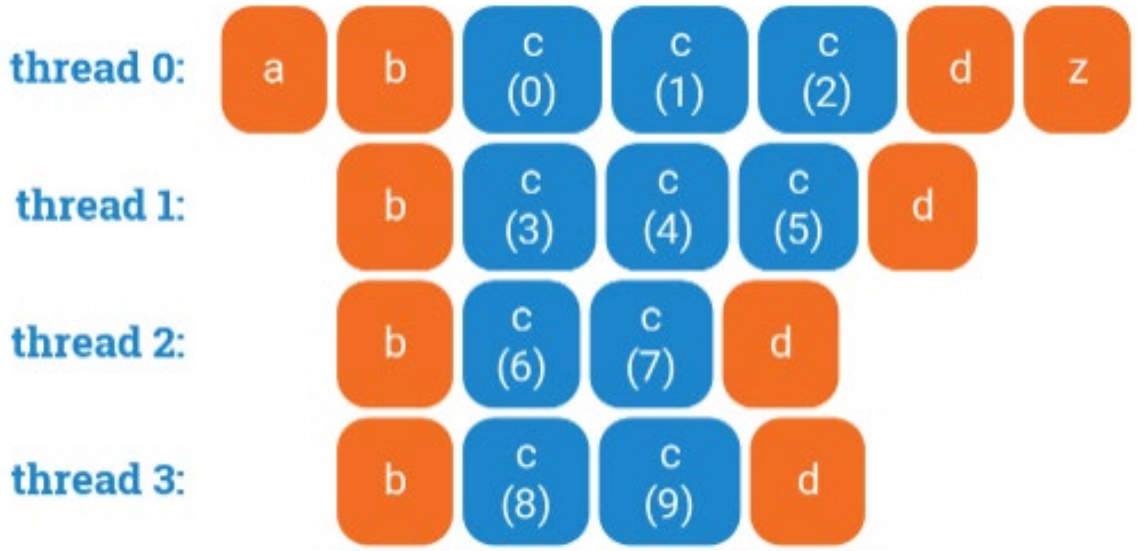
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



Każdy obszar równoległy zakończony jest (niejawną) barierą synchronizacyjną. Również po każdej pętli **opm for** jest punkt synchronizacji. Na rysunku: funkcja d() nie zostanie wykonana aż do zakończenia pracy przez wszystkie wątki wewnątrz pętli.

Pętla for – bez czekania (nowait)

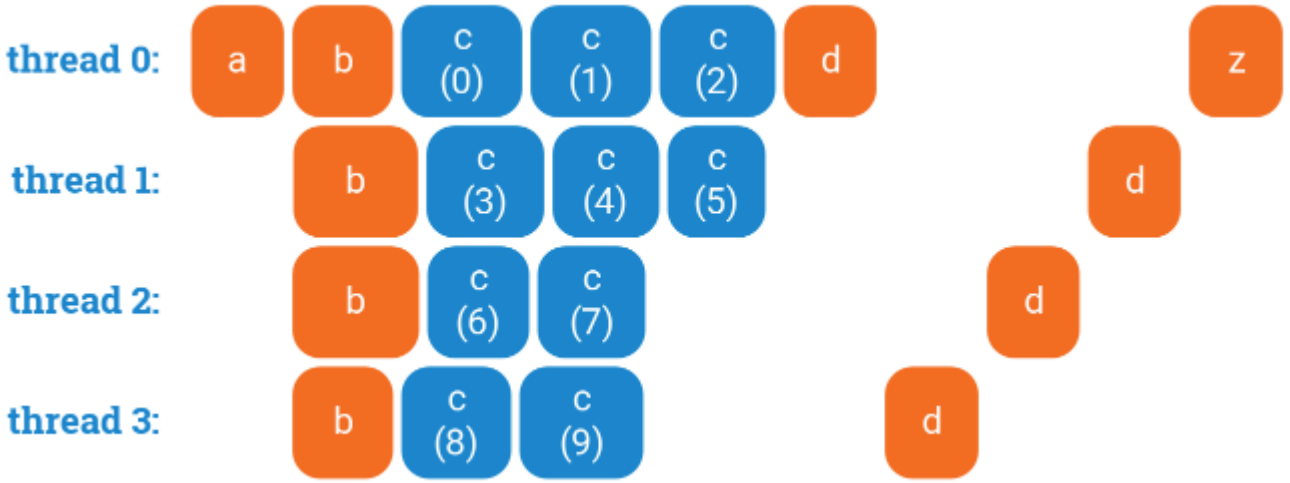
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



Jeśli synchronizacja po zakończeniu pętli **opm for** nie jest konieczna, można ją wyłączyć klauzulą **nowait**. Na rysunku: funkcja d() zostanie wykonana od razu po zakończeniu pracy (w pętli) przez każdy z wątków.

Pętla for – interakcja z sekcją krytyczną

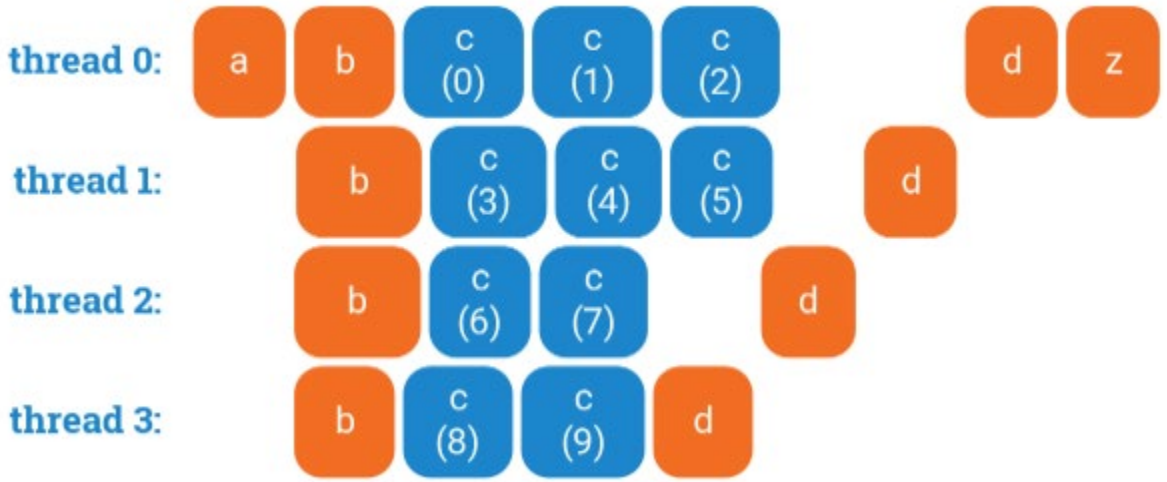
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    #pragma omp critical
    {
        d();
    }
}
z();
```



Jeśli za pętlą **opm for** znajduje się sekcja krytyczna, to najpierw nastąpi synchronizacja na końcu pętli, by potem kolejne wątki mogły wykonać pojedynczo sekcję krytyczną (tutaj funkcję d()).

Pętla for – (nowait) i sekcja krytyczna

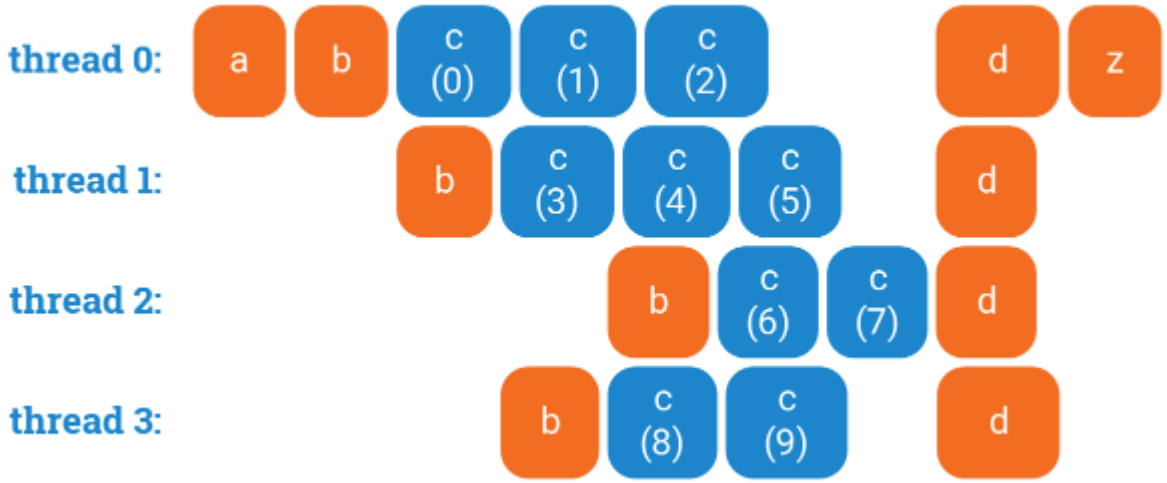
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    #pragma omp critical
    {
        d();
    }
}
z();
```



Można wyłączyć synchronizację za pętlą **opm for**, wtedy wątek od razu zaczyna realizować część krytyczną – ochrona tej części jest nadal zapewniona (tylko jedna funkcja d() w danej chwili). Ma to sens, gdy funkcja d() realizuje np. aktualizację jakiejś zmiennej globalnej, wyliczoną przez dany wątek.

Pętla for – sekcja krytyczna przed pętlą

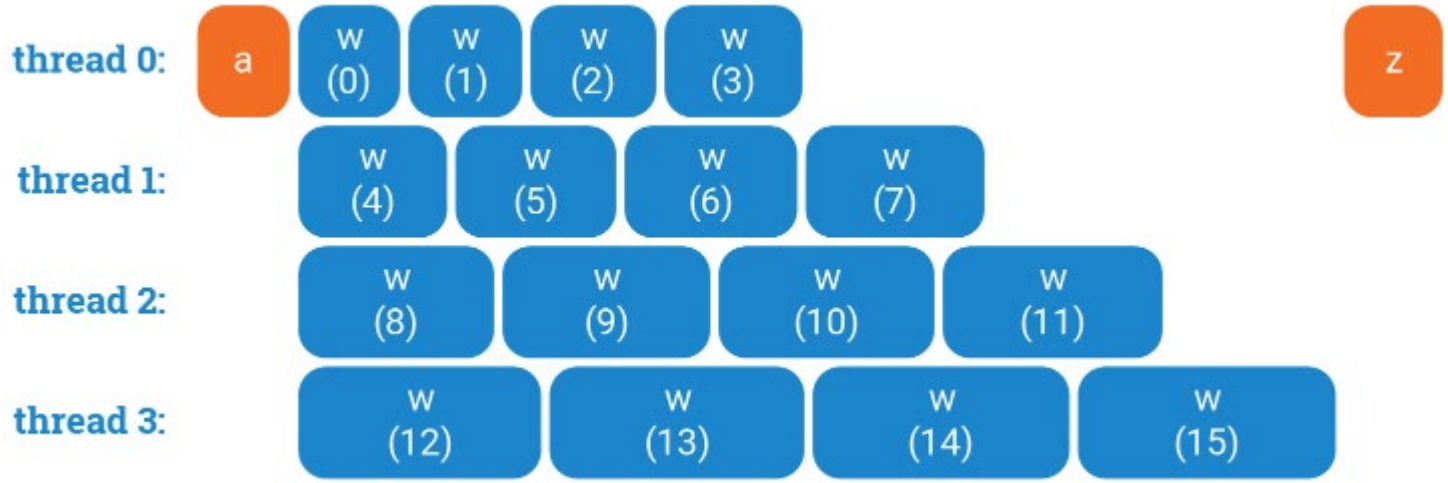
```
a();
#pragma omp parallel
{
    #pragma omp critical
    {
        b();
    }
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



Przed pętlą **opm for** nie ma żadnego punktu synchronizacji – zatem po wykonaniu funkcji b() zostanie od razu wykonana część pętli delegowana do danego wątku.

Pętla for – warianty schedule

```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



Przypadek: w pętli wywołana jest funkcja w(i), której czas wykonania rośnie wraz z indeksem i. Pętla **opm for** uruchomiona z opcją domyślną, dzieli kolejne części pętli w postaci proporcjonalnych bloków. Np. pierwszy wątek wykona funkcję w(0), w(1), w(2), w(3), kolejny wykona w(4), w(5), w(6), w(7) itd. W tym przykładzie wątek 0 zakończy się najwcześniej, a wątek 3 najpóźniej.

Pętla for – domyślne

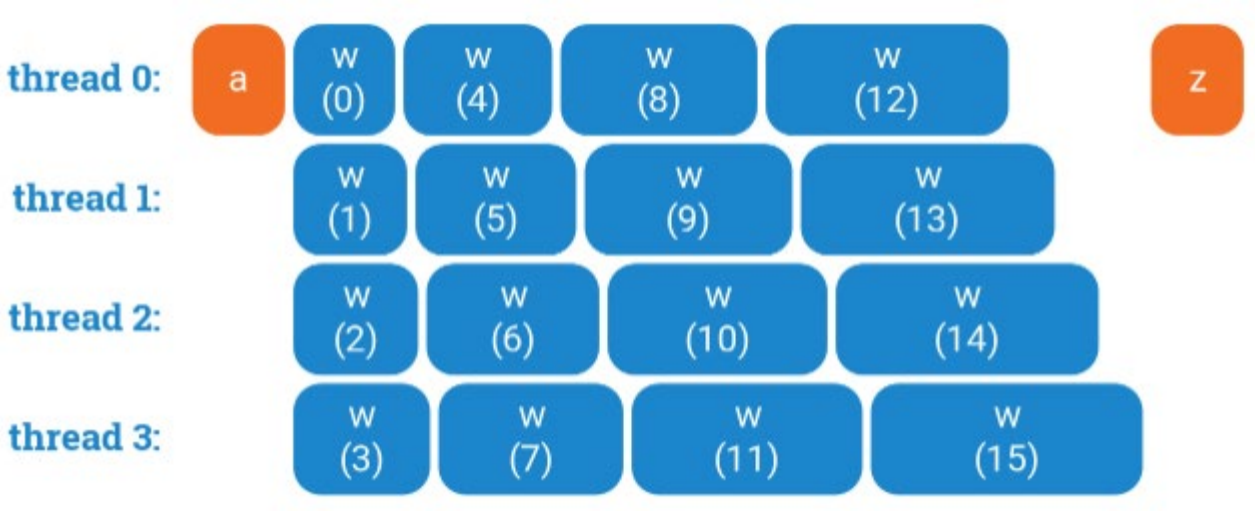
```
#include <omp.h>
#include <iostream>
using namespace std;

void w(long n) {
    long sum = 0;
    cout << "i = " << (n/100000001)
         << " thread id = " << omp_get_thread_num() << endl;
    for (long i = 0; i < 20*n; ++i) sum += i;
}

int main()
{
    const long max = 16, factor = 100000001;
    #pragma omp parallel for num_threads(4)
        for (int i=0; i<max; ++i) w(i*factor);
}
```

Program testujący:
wykonamy pętlę na 4 wątkach. Funkcja **w(long n)** wywoływana jest w pętli 16. razy, z argumentem rosnącym wraz z indeksem iteracji **i**.

Najwolniej (najdłuższe obliczenia) wykona się wątek nr 3. Czas całkowity: 32 sek.

Pętla for – schedule(static,1)

```
a();
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



Pętla **opm for** uruchomiona z klauzulą schedule(static,1) dokonuje przypisania kolejnych wywołać funkcji **w** na zasadzie cyklicznej rotacji (iteracja 0 do wątku 0, iteracja 1 do wątku 1, itd.). W tym przypadku rozmiar (chunk) 1 oznacza, że jedna iteracja delegowana jest do wątku, po czym następuje przełączenie na kolejny wątek.

→ na podstawie poprzedniego kodu przetestuj wariant schedule(static,1)

Pętla for – schedule(static,1)

```
#include <omp.h>
#include <iostream>
using namespace std;

void w(long n) {
    long sum = 0;
    cout << "i = " << (n/100000001)
         << " thread id = " << omp_get_thread_num() << endl;
    for (long i = 0; i < 20*n; ++i) sum += i;
}

int main()
{
    const long max = 16, factor = 100000001;
    #pragma omp parallel for schedule(static,1) num_threads(4)
        for (int i=0; i<max; ++i) w(i*factor);
}
```

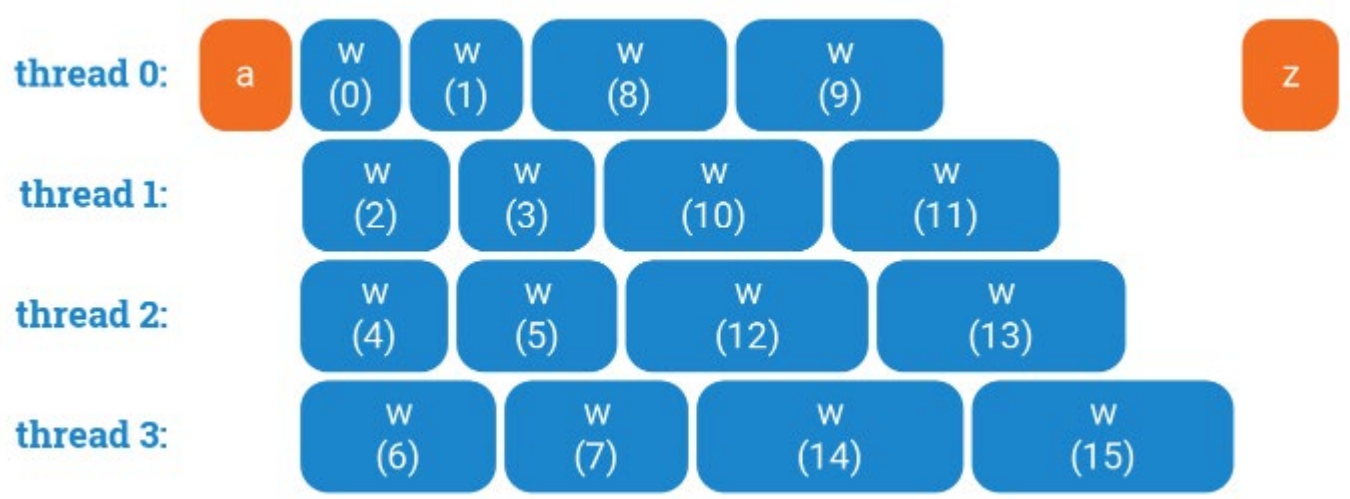
```
i = i = 0 thread id = 30 thread id = 3
i = 4 thread id = 0
i = 2 thread id = 2

i = 1 thread id = 1
i = 5 thread id = 1
i = 6 thread id = 2
i = 7 thread id = 3
i = 8 thread id = 0
i = 9 thread id = 1
i = 10 thread id = 2
i = 11 thread id = 3
i = 12 thread id = 0
i = 13 thread id = 1
i = 14 thread id = 2
i = 15 thread id = 3

real    0m20.663s
user    1m9.416s
sys     0m0.112s
```

Czas całkowity: 21 sek.

(lepiej niż domyślnie – 32 sek.)

Pętla for – schedule(static,2)

```
a();
#pragma omp parallel for schedule(static,2)
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



Pętla **opm for** uruchomiona z klauzulą schedule(static,2) deleguje po dwa kolejne wywołania funkcji **w** na zasadzie cyklicznej rotacji (iteracja 0 i 1 do wątku 0, iteracja 2 i 3 do wątku 1, itd.).

→ na podstawie poprzedniego kodu przetestuj wariant schedule(static,2)

Pętla for – schedule(static,2)

```
#include <omp.h>
#include <iostream>
using namespace std;

void w(long n) {
    long sum = 0;
    cout << "i = " << (n/100000001)
         << " thread id = " << omp_get_thread_num() << endl;
    for (long i = 0; i < 20*n; ++i) sum += i;
}

int main()
{
    const long max = 16, factor = 100000001;
    #pragma omp parallel for schedule(static,2) num_threads(4)
        for (int i=0; i<max; ++i) w(i*factor);
}
```

```
i = i = 0 thread id = 02 thread id = 1
i = 1 thread id = 0

i = i = 46 thread id = 2 thread id =
3
i = 8 thread id = 0
i = 3 thread id = 1
i = 5 thread id = 2
i = 10 thread id = 1
i = 7 thread id = 3
i = 9 thread id = 0
i = 12 thread id = 2
i = 14 thread id = 3
i = 11 thread id = 1
i = 13 thread id = 2
i = 15 thread id = 3

real    0m24.075s
user    1m8.904s
sys     0m0.004s
```

Czego należy się spodziewać w przypadku **schedule(static,4)** ?
A w przypadku **schedule(static,8)** ?
albo nawet **schedule(static,16)** ?

Czas całkowity: 24 sek.
(gorzej niż poprzednio!)

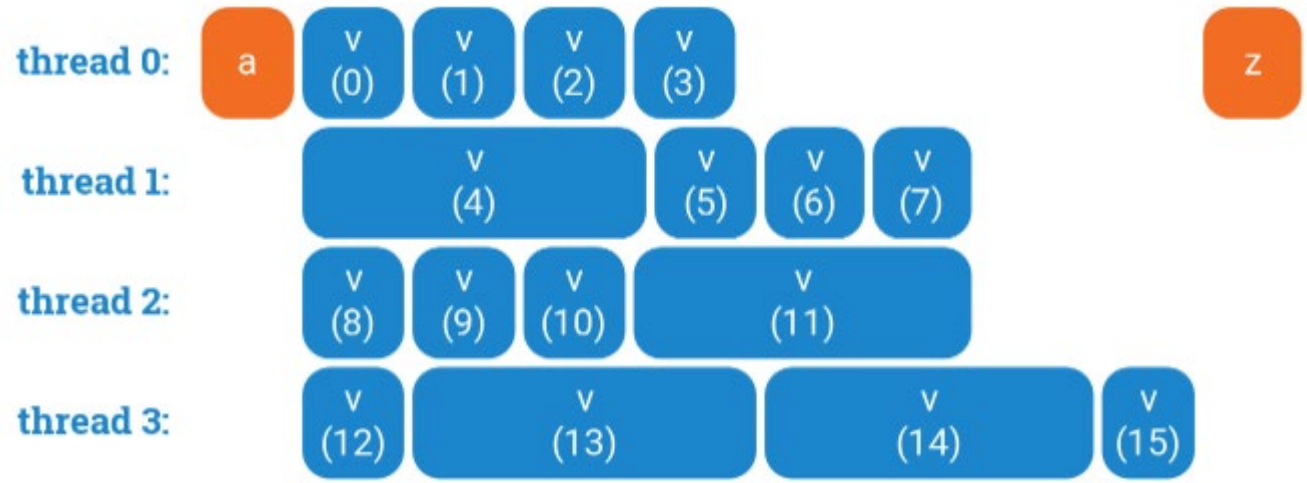
Pętla for – domyślna, schedule(static)

Rozdzielanie iteracji pętli na wątki w wersji domyślnej i statycznej jest bardzo wydajne: nie ma potrzeby komunikacji między wątkami. Kiedy pętla się rozpocznie, każdy wątek natychmiast będzie wiedział, które iteracje pętli wykona (decyzja podjęta podczas kompilacji). Jedyna część synchronizacji znajduje się na końcu całej pętli, aby zakończyć wszystkie wątki.

Gdy obciążenie poszczególnych iteracji jest zmienne, np. są iteracje, które trwają znacznie dłużej, w powyższych strategiach może się okazać, że niektóre wątki pracują znacznie dłużej.

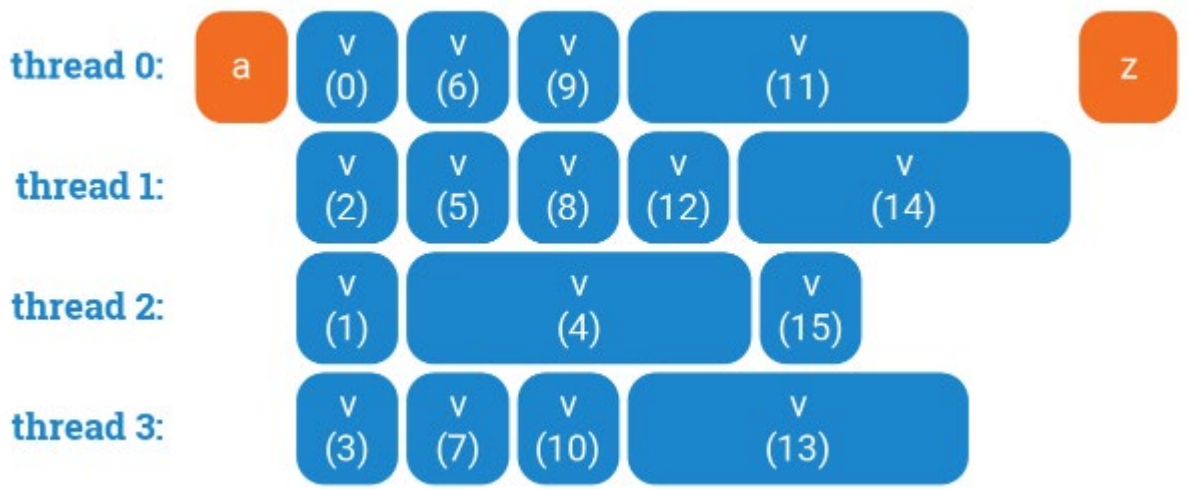
```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```

Funkcja **v** jest nieprzewidywalna.



Pętla for – schedule(dynamic,1)

```
a();
#pragma omp parallel for schedule(dynamic,1)
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```



W dynamicznym szeregowaniu każdy wątek wykonuje jedną iterację, przetwarza ją, a następnie sprawdza, jaka jest następna iteracja, która nie jest obecnie przetwarzana przez nikogo. W ten sposób nigdy nie stanie się tak, że jeden wątek kończy się, podczas gdy inne wątki mają jeszcze dużo pracy do wykonania.

→ na podstawie poprzedniego kodu przetestuj wariant schedule(dynamic,1)

Pętla for – schedule(dynamic,1)

```
#include <omp.h>
#include <iostream>
using namespace std;

void w(long n) {
    long sum = 0;
    cout << "i = " << (n/100000001)
         << " thread id = " << omp_get_thread_num() << endl;
    for (long i = 0; i < 20*n; ++i) sum += i;
}

int main()
{
    const long max = 16, factor = 100000001;
    #pragma omp parallel for schedule(dynamic,1) num_threads(4)
        for (int i=0; i<max; ++i) w(i*factor);
}
```

```
i = 1 thread id = 0
i = 2 thread id = 3
i = 3 thread id = 1
i = 0 thread id = 2
i = 4 thread id = 2
i = 5 thread id = 0
i = 6 thread id = 3
i = 7 thread id = 1
i = 8 thread id = 2
i = 9 thread id = 0
i = 10 thread id = 3
i = 11 thread id = 1
i = 12 thread id = 2
i = 13 thread id = 0
i = 14 thread id = 3
i = 15 thread id = 1

real    0m20.713s
user    1m8.840s
sys     0m0.000s
```

→ przetestuj wariant schedule(dynamic,2)

Czas całkowity: 21 sek.
(podobnie do static,1)

Pętla for – schedule(dynamic,2)

```
#include <omp.h>
#include <iostream>
using namespace std;

void w(long n) {
    long sum = 0;
    cout << "i = " << (n/100000001)
         << " thread id = " << omp_get_thread_num() << endl;
    for (long i = 0; i < 20*n; ++i) sum += i;
}

int main()
{
    const long max = 16, factor = 100000001;
    #pragma omp parallel for schedule(dynamic,2) num_threads(4)
        for (int i=0; i<max; ++i) w(i*factor);
}
```

```
i = i = 2 thread id = 04
i = 6 thread id = 2
 thread id = 3
i = 0 thread id = 1
i = 1 thread id = 1
i = 8 thread id = 1
i = 3 thread id = 0
i = 5 thread id = 3
i = 10 thread id = 0
i = 7 thread id = 2
i = 9 thread id = 1
i = 12 thread id = 3
i = 14 thread id = 2
i = 11 thread id = 0
i = 13 thread id = 3
i = 15 thread id = 2

real    0m24.302s
user    1m9.624s
sys     0m0.100s
```

Czas całkowity: 24 sek.

(podobnie do static,2)

Pętla for – schedule(dynamic)

Planowanie **dynamiczne** jest **kosztowne**: istnieje pewna komunikacja między wątkami po każdej iteracji pętli! Zwiększenie wielkości porcji (liczba "1" w dyrektywie) może pomóc w znalezieniu lepszego kompromisu między zbilansowanym nakładem pracy a kosztami koordynacji.

Warto również zauważyć, że planowanie dynamiczne niekoniecznie zapewnia optymalny harmonogram. OpenMP nie może przewidzieć przyszłości: przypisywanie pętli iteracji do wątków odbywa się na zasadzie „chciwego” podejścia (kolejny wątek zabiera bez analizy następną wolną iterację).

Co w przypadku schedule(guided) ?
schedule(guided,1) - 21 sek.
schedule(guided,2) - 25 sek.

Co w przypadku schedule(auto) ?
schedule(auto) - 31 sek. (niedobrze!)

schedule(runtime)
Bardzo wygodne do testowania, ponieważ zmiana opcji wykonania (jeśli ustawiona za pomocą zmiennej OMP_SCHEDULE nie wymaga rekompilacji kodu).

Pętla for – testy (losowa funkcja)

Wewnątrz funkcji **v** jest losowana liczba całkowita w zakresie **i** = 1-30, emulująca zmienną długość „obliczeń”
(argument nie zależy od indeksu pętli for, tylko od wartości wylosowanej).

```
#include <omp.h>
#include <iostream>
#include <random>
using namespace std;

void v(long n) {
    random_device rd;
    uniform_int_distribution<int> dist(1,20);
    long sum = 0; long los = dist(rd);
    cout << "i = " << los
        << " thread id = " << omp_get_thread_num() << endl;
    for (long i = 0; i < los*n; ++i) sum += i;
}

int main()
{
    const long max = 16, factor = 100000001;
    #pragma omp parallel for num_threads(4)
        for (int i=0; i<max; ++i) v(factor);
}
```

```
i = 19 thread id = 0
i = 14 thread id = 2
i = 10 thread id = 1
i = 13 thread id = 3
i = 13 thread id = 1
i = 9 thread id = 3
i = 8 thread id = 2
i = 9 thread id = 0
i = 9 thread id = 2
i = 14 thread id = 3
i = 19 thread id = 1
i = 2 thread id = 0
i = 7 thread id = 0
i = 18 thread id = 2
i = 15 thread id = 3
i = 11 thread id = 1

real    0m1.675s
user    0m5.976s
sys     0m0.004s
```

Czas całkowity: 1,68 sek.
(wersja sekwencyjna: 6 sek)

Pętla for – testy (losowa funkcja)

Wyniki dla różnych wersji optymalizacji **omp for**

schedule(static,1) – 1,86 s.
schedule(static,2) – 1,65 s.
schedule(static,4) – 1,65 s.
schedule(dynamic,1) – 1,40 s.
schedule(dynamic,2) – 1,80 s.
schedule(dynamic,4) – 2,01 s.
schedule(guided,1) – 1,17 s.
schedule(guided,2) – 1,46 s.
schedule(guided,4) – 1,46 s.
schedule(auto) – 1,76 s.

Konkluzje

Spory rozrzut w czasie wykonania – nigdy nie wiemy ile będzie trwać wykonanie iteracji.

Wyniki nie do przewidzenia – każdą sytuację trzeba **testować**!