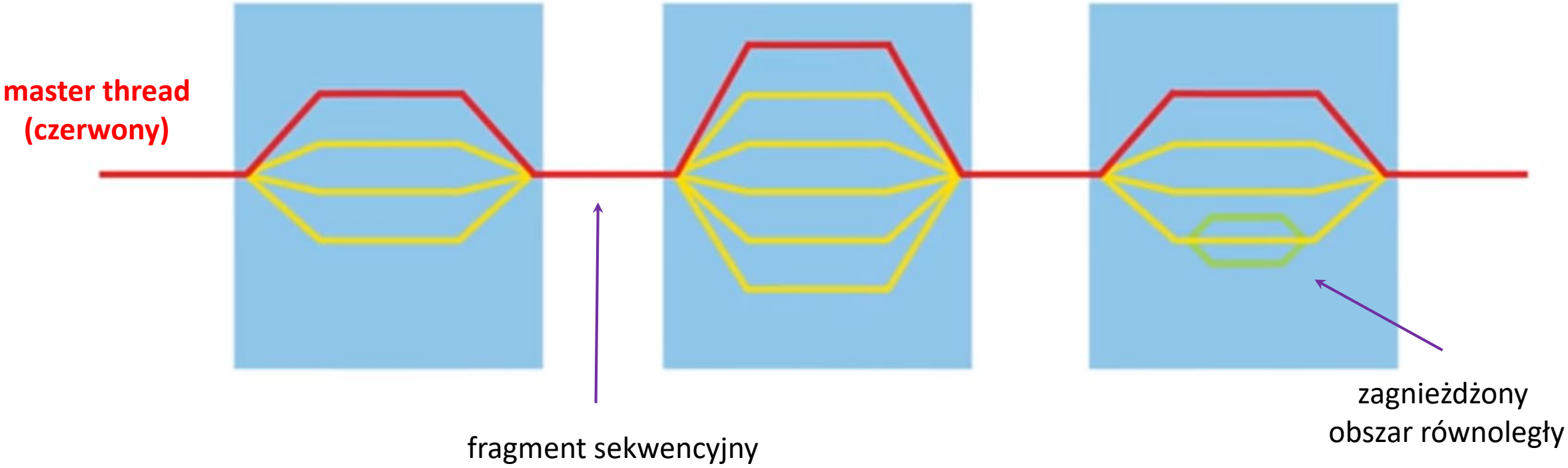
Wątki w OpenMP (model fork-join)



- Wątek master (czerwona) rozdziela się na grupę (team) wątków tam, gdzie to możliwe
- Podział na wątki dodawany stopniowo, tj. sekwencyjny program przekształca się w program równoległy.

Tworzenie wątku: obszary równoległe

W OpenMP tworzysz wątki z konstrukcją równoległą

Przykład: aby utworzyć 4-wątkowy równoległy obszar:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID,A);
}
```

funkcja wykonawcza żądająca wykonane 4 wątków

każdy wątek wykonuje kopię kodu w bloku strukturalnym (poniżej #pragma)

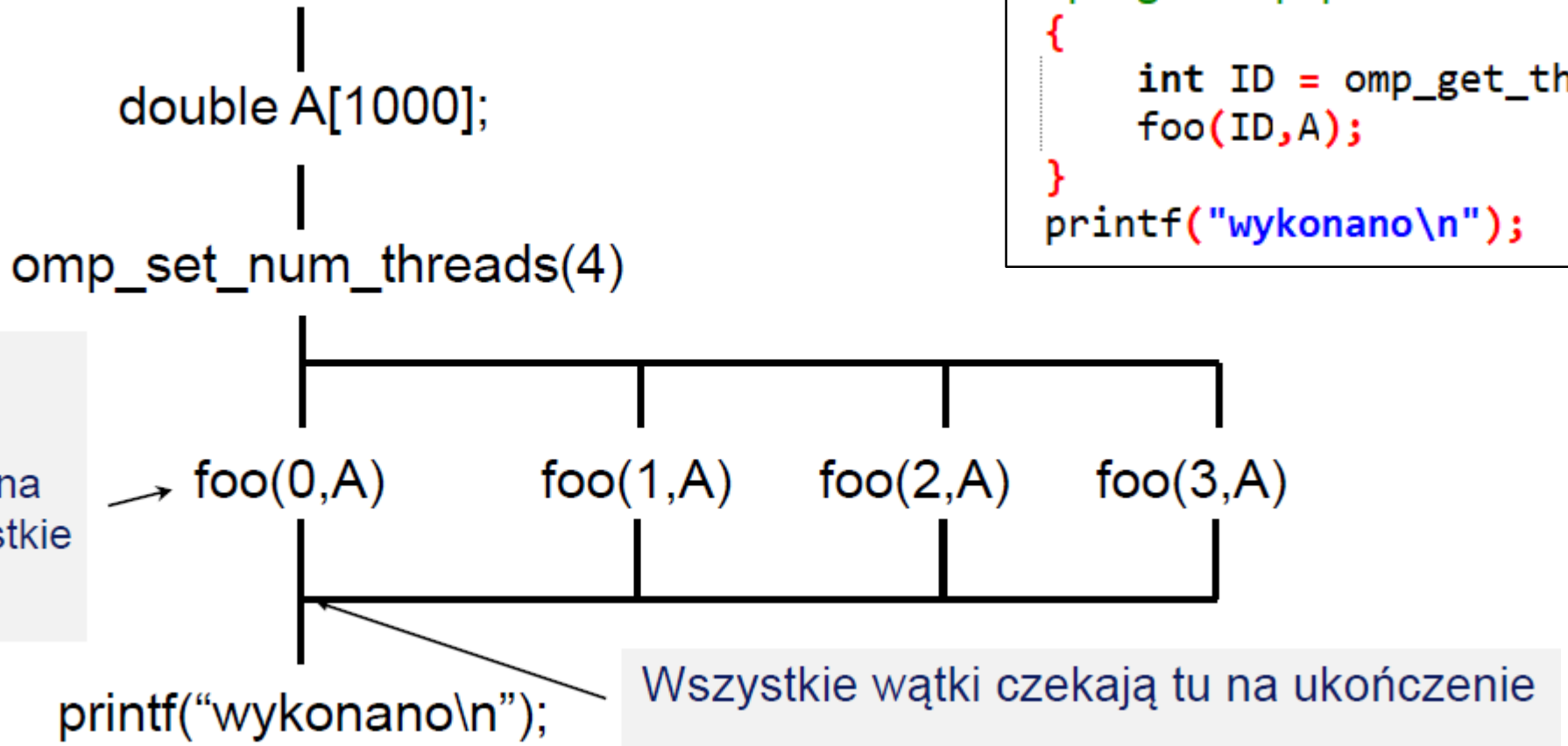
funkcja wykonawcza zwracająca numer wątku

Każdy wątek wykonuje foo(ID,A) dla ID = 0 do 3

Tworzenie wątku: obszary równoległe

Każdy wątek wykonuje ten sam kod.

```
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    foo(ID,A);
}
printf("wykonano\n");
```



Pojedyncza kopia A jest współdzielona przez wszystkie wątki

Wszystkie wątki czekają tu na ukończenie

Tworzenie wątku: co robi kompilator

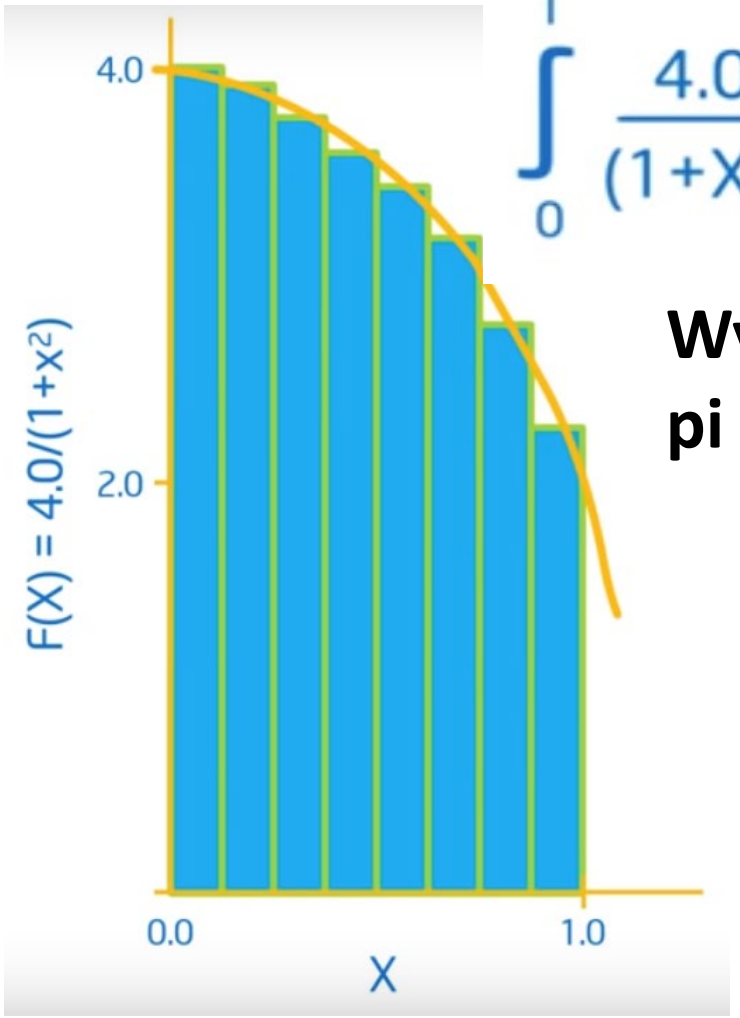
```
#pragma omp parallel num_threads(4)
{
    foobar();
}
```

```
void thunk()
{
    foobar();
}

pthread_t tid[4];
for (int i=1; i<4; ++i)
    pthread_create(&tid[i],0,thunk,0);
thunk();

for (int i=1; i<4; ++i)
    pthread_join( tid[i] );
```

- Kompilator OpenMP generuje kod (jak po prawej stronie slajdu), biorąc pod uwagę pragma OpenMP, jak ta u góry
- OpenMP korzysta z puli wątków. Wątek nadrzędy to wątek 0.
- Tworzone są tylko trzy wątki, ponieważ ostatnia sekcja równoległa zostanie wywołana z wątku nadrzędnego.

Ćwiczenie: całka z funkcji

Całka z funkcji (pole powierzchni)



$$\int_0^1 \frac{4.0}{(1+X^2)}\,dx = \pi$$

Wynik:
pi = 3.14159

Wiemy, że:

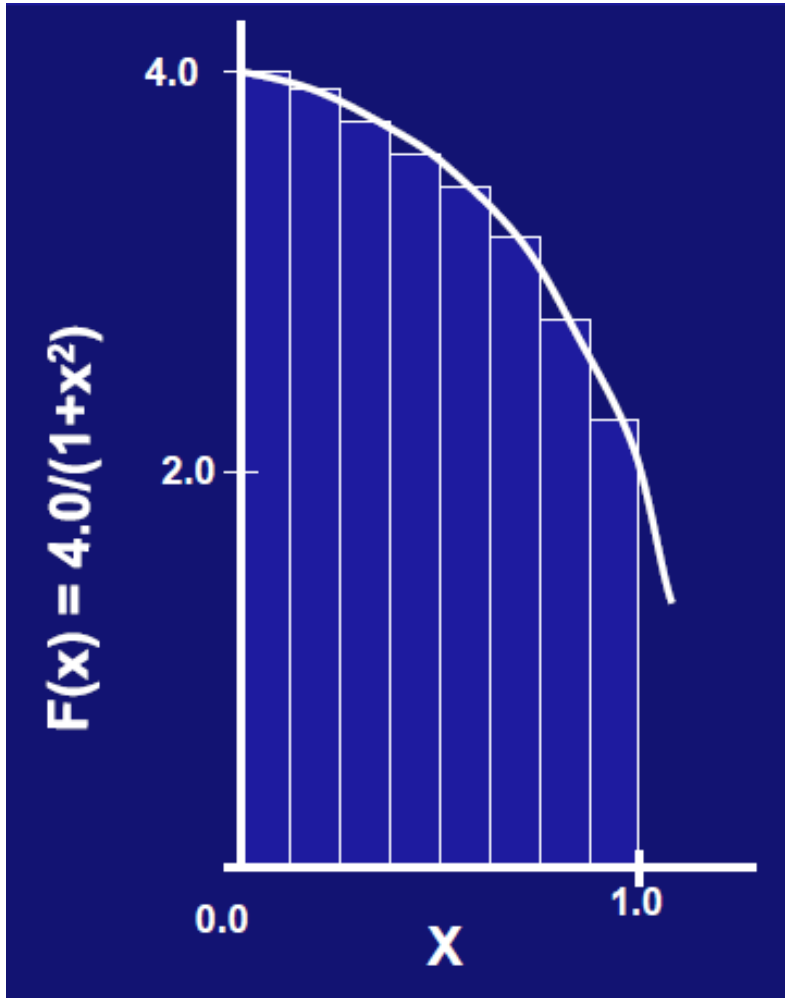
$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

Można przybliżyć wartość całki poprzez sumę pól prostokątów

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

gdzie każdy prostokąt ma szerokość Δx oraz wysokość F(x_i) w środku przedziału *i*

Ćwiczenie: całka z funkcji



pi = 3.14159

Numeryczne obliczenie pola powierzchni:

```
#include <iostream>
using namespace std;

static long num_steps = 100000;
double step;
int main()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;
    for (i=0; i<num_steps; ++i) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    cout << "PI = " << pi << endl;
}
```

Ćwiczenie: całka z funkcji

```
int omp_get_num_threads();
```
Number of threads in the team

```
int omp_get_thread_num();
```
Thread ID or rank

```
double omp_get_wtime();
```
Time in seconds since a fixed point in the past

- Za pomocą **#pragma omp parallel** napisać zrównoleglоną wersję programu
- Zwrócić uwagę na prywatne / współdzielone zmienne
- Wykorzystać również funkcje biblioteczne

- liczba wątków w grupie
- ID wątku
- czas w sekundach od oznaczonego punktu w czasie

Kod

g++ -fopenmp main.cc -o program

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

static long num_steps = 300000000; // w C++14 można: 300'000'000;
const unsigned num_threads = 2;
double step;
int main()
{
    int i, nthreads; double pi, sum[ num_threads ];
    step = 1.0 / (double)num_steps;
    omp_set_num_threads( num_threads );
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id==0) nthreads = nthrds;
        for (i=id, sum[id]=0.0; i<num_steps; i+=nthrds) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0 + x*x);
        }
    }
    for (i=0, pi=0.0; i<nthreads; ++i) pi += step * sum[i];
    cout << "PI = " << pi << endl;
}
```

Zamiana skalarnej zmiennej sum na wersję tablicową (o rozmiarze liczby przewidywanych wątków), celem uniknięcia ”data race”

Jeden (aby nie było ”data race”) wątek (np. master thread) zapisuje informację o rzeczywistej liczbie wątków

Typowa sztuczka w programach SPMD do tworzenia cyklicznej dystrybucji iteracji pętli

SPMD = **S**ingle **P**rogram **M**ultiple **D**ata

Strategia algorytmu

Wzorzec **SPMD** (**S**ingle **P**rogram **M**ultiple **D**ata)

- Uruchom ten sam program na P elementach przetwarzanych równolegle, gdzie P może być dowolnie duże
- Użyj identyfikatora od 0 do (P-1), aby wybrać między zestawem zadań i zarządzać dowolnymi współdzielonymi strukturami danych

Ten wzorzec jest bardzo ogólny i jest chyba najczęściej używaną strategią w historii programowania równoległego.

Wyniki (skalowanie z liczbą wątków)

Przykładowo, na maszynie klasy Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz, otrzymano na oryginalnym programie (bez podziału na wątki), przy liczbie kroków 300000000, czas wykonania 1,52 sekund.

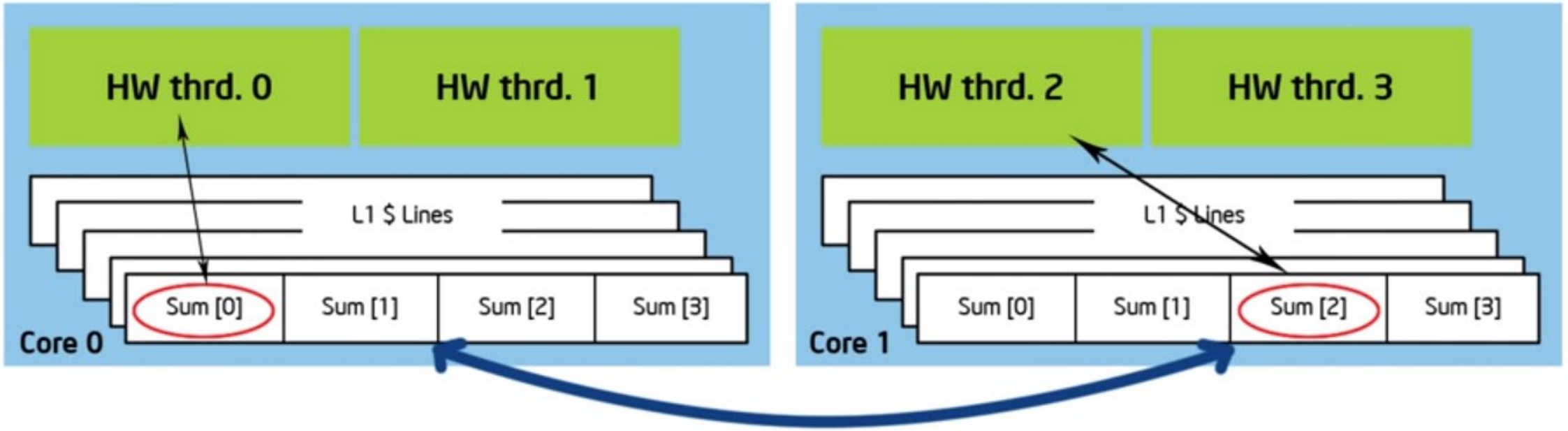
Po podziale na wątki

Thread	SPMD
1	1,72
2	3,90
4	4,22
40	2,53

Twoje wyniki? Jakie spostrzeżenia?

Dlaczego tak słabe wyniki?

Jeśli niezależne elementy danych (jak np. części naszej tablicy, aktualizowanej przez różne wątki) znajdują się w tym samym bloku danych pamięci podręcznej (**cache line**), każda ich aktualizacja spowoduje przeskakiwanie w tę i na powrót pomiędzy wątkami.
To jest przykład tzw. fałszywego współdzielenia (**false sharing**).



Rozwiązanie (false sharing)

- Jeśli zamienimy wielkość skalarną (czyli jedną zmienną) na macierz (tablicę) wspierającą realizację programu **SPMD**, elementy tej tablicy są rozmieszczone w pamięci w sposób ciągły (jeden za drugim), a zatem współdzielą ten sam blok pamięci podręcznej (*cache line*). Skutkuje to bardzo niską skalowalnością względem obliczeń na wielu wątkach, lub wręcz gorszymi wynikami niż jednowątkowy program!
- **Rozwiązanie** (nieeleganckie): Tak „powiększyć” tablice, które będą aktualizowane, aby ich poszczególne części (aktualizowane przez wątki) nie znajdowały się w tym samym bloku pamięci. Można to osiągnąć dodać drugi wymiar do pierwotnej tablicy: double sum [num_threads];
  Rozmiar typu double to 8 bajtów. Utworzymy sztucznie tablicę 2-wymiarową.

O ile i jak „powiększyć” tablicę?

- **Rozmiar bloku pamięci zależy od konkretnej architektury**. W systemach linuxowych, cache line size można odczytać na kilka sposobów.
- W katalogu: /sys/devices/system/cpu/cpu0/cache/ są podkatalogi dla każdego poziomu cache, np. index0 index1 index2 index3
  Każdy z nich zawiera pliki (coherency_line_size, level, number_of_sets, physical_line_partition, shared_cpu_list, shared_cpu_map, size, type, ways_of_associativity), z coherency_line_size można odczytać rozmiar: **64**
- Inny sposób (linia komend): getconf LEVEL1_DCACHE_LINESIZE
- Inny sposób (parametry procesora): cat /proc/cpuinfo
  cache_alignment : **64**
- Jeśli dodamy drugi rozmiar o wielkości **8** × (double) = **64** bajty, to mamy gwarancję, że nawet dla 2. wątków, kolejne pozycje w pierwszym rozmiarze tablicy, aktualizowane przez wątki, będą w różnych blokach cache.

Kod 2

g++ -fopenmp main.cc -o program

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

static long num_steps = 300000000; // w C++14 można: 300'000'000;
const unsigned PAD = 8; // dla 64-bajtowej pamięci cache poziomu L1
const unsigned num_threads = 2;
double step;
int main()
{
    int i, nthreads; double pi, sum[ num_threads ][ PAD ];
    step = 1.0 / (double)num_steps;
    omp_set_num_threads( num_threads );
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id==0) nthreads = nthrds;
        for (i=id, sum[id][0]=0.0; i<num_steps; i+=nthrds) {
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0 + x*x);
        }
    }
    for (i=0, pi=0.0; i<nthreads; ++i) pi += step * sum[i][0];
    cout << "PI = " << pi << endl;
}
```

Teraz tablica sum jest dwuwymiarowa, każda kolejna pozycja dla indeksu w rozmiarze num_threads jest w odległości PAD × double (czyli tu 64 bajty)

W każdym miejscu kodu zamiast sum[i] używamy teraz dodatkowo drugi wymiar, np. indeksem 0, sum[i][0]

Wyniki (powiększona tablica)

Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz, bez podziału na wątki, przy liczbie kroków 300000000, czas wykonania **1,52** sekund.

Po podziale na wątki

Thread	SPMD	SPMD padded
1	1,72	1,70
2	3,90	0,95
4	4,22	0,54
40	2,53	0,15