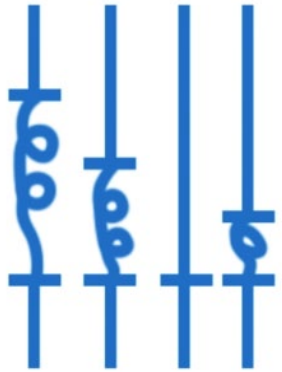
OpenMP: współdziałanie wątków

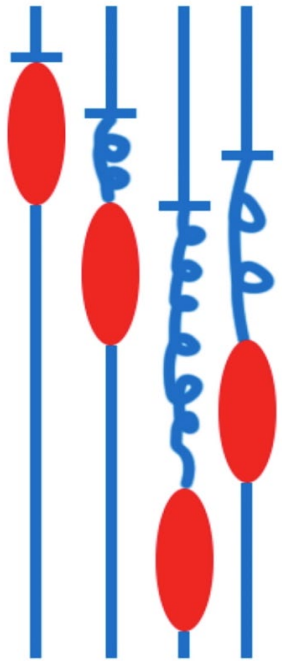
Współpraca wątków

- OpenMP jest oparte na modelu działania wielowątkowego ze współdzielonym adresowaniem
  - Wątki komunikują się ze sobą poprzez współdzielone wartości
- Niezaplanowane współdzielenie danych spowoduje ”data race”
  - Przyczyny: modyfikacje danych wyjściowych przez asynchronicznie działające różne wątki
- Kontrola „wyścigu do danych”
  - Używać synchronizacji celem zapobiegania konfliktom.
- Synchronizacja jest kosztowna, zatem:
  - Sprawdzić dostęp do danych celem minimalizacji potrzeb synchronizacyjnych.

Rodzaje synchronizacji



Bariera (barrier) – każdy wątek czeka w miejscu bariery, aż wszystkie wątki zakończą działanie



Wzajemne wykluczenie (mutual exclusion) – zdefiniowanie bloku kodu, który w danej chwili czasu może wykonywać tylko jeden wątek

Synchronizacja (rodzaje)

Synchronizacja wysokiego poziomu:

- critical
- atomic
- barrier
- ordered

Synchronizacja służy do nałożenia ograniczeń co do kolejności oraz ochrony dostępu do danych współdzielonych

Synchronizacja niskiego poziomu:

- flush
- locks (proste i zagnieżdżone)

Rodzaje synchronizacji

Bariera (barrier) – każdy wątek czeka, aż wszystkie dotrą do miejsca oznaczonego jako bariera

```
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    A[ID] = big_calc1( ID ); // A duża tablica

#pragma omp barrier

    B[ID] = big_calc2( ID );

}
```

Wzajemne wykluczenie (mutual exclusion) – tylko jeden wątek może w danej chwili wejść do obszaru krytycznego

```
float res;
#pragma omp parallel
{
    float b;
    int id = omp_get_thread_num();
    int nthrds = omp_get_num_threads();
    for ( int i=id; i<niters; i+=nthrds ) {
        B = big_job(i);
    #pragma omp critical
        res += consume(B);
    }
}
```

Sekcja krytyczna może być w { }
Nie należy nadużywać, bo spowoduje to serializację obliczeń i utratę zalet obliczeń równoległych

Wątki czekają na swoją kolej – tylko jeden w danej chwili czasu wykonuje funkcję consume

Rodzaje synchronizacji

Atomowa (atomic) – rodzaj operacji typu „wzajemnego wykluczania” dla niewielkich operacji (najczęściej aktualizacji jakiejś wartości), np. wspieranych sprzętowo.

```
#pragma omp parallel
{
    double tmp, B;
    B = doIt();
    tmp = very_big_calc(B);

    #pragma omp atomic
        X += tmp;
}
```

Wyrażenia wewnątrz bloku **atomic** muszą być typu:
X *operator_binarny*= wyrażenie
X++
++X
X--
--X

X musi być skalarną *lewą*-wartością zaś *operator_binarny* musi być nieprzeciążonym (wbudowanym) operatorem języka.

Rodzaje synchronizacji

Uporządkowana (ordered) – wymusza wykonanie oznaczonej instrukcji/bloku wewnątrz pętli dokładnie w takiej kolejności, w jakiej byłyby wykonane w programie sekwencyjnym.

```
#pragma omp parallel private(tmp)
{
  #pragma omp for oredered reduction(+:wynik)
  for (i=0; i<N; ++i) {
    tmp = obliczenia(i);
    #pragma ordered
    wynik += przelicz(tmp);
  }
}
```

W danym momencie tylko jeden wątek może wykonywać oznaczoną instrukcję, gdy pozostałe czekają.

Pętla zawierająca blok z klauzulą ordered musi również mieć tę klauzulę.

g++ -fopenmp main.cc -o program

Kod

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

static long num_steps = 300000000;
const unsigned num_threads = 4;
int main()
{
    double pi = 0.0, nthreads;
    double step = 1.0 / (double)num_steps;
    omp_set_num_threads( num_threads );
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x, sum;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id==0) nthreads = nthrds;
        for (i=id, sum=0.0; i<num_steps; i+=nthreads) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0 + x*x);
        }
        #pragma omp critical
        pi += sum * step;
    }
    cout << "PI = " << pi << endl;
}
```

sum – teraz jest lokalną zmienną każdego wątku, akumulującą sumy cząstkowe

nie ma tablicy, nie ma fałszywego współdzielenia (false sharing)

sum nie istnieje poza blokiem równoległym, więc końcowa wartość musi być wyliczona wewnątrz bloku: musimy zabezpieczyć sumowanie **pi** poprzez blok krytyczny

Wyniki (critical section)

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

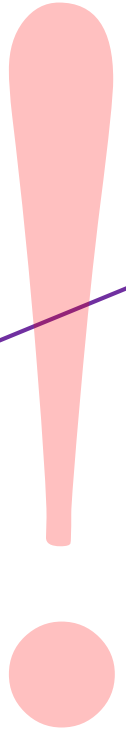
Thread	SPMD padded	SPMD critical
1	1,70	1,80
2	0,95	1,02
4	0,54	0,55
40	0,15	0,15

Kod 2

g++ -fopenmp main.cc -o program

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

static long num_steps = 300000000;
const unsigned num_threads = 40;
int main()
{
    double pi = 0.0, nthreads;
    double 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; i<num_steps; i+=nthreads) {
            x = (i+0.5)*step;
        #pragma omp critical
            pi += 4.0/(1.0 + x*x);
        }
    }
    pi *= step;
    cout << "PI = " << pi << endl;
}
```



sum – nie ma w ogóle

przeniesienie części krytycznej do pętli

ponieważ obliczenie **x** jest bardzo szybkie, umieszczenie sekcji krytycznej praktycznie prowadzi do serializacji obliczeń i katastrofalnego czasu obliczeń (40 wątków: prawie 3 minuty)!

Kod 3

g++ -fopenmp main.cc -o program

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

static long num_steps = 300000000;
const unsigned num_threads = 40;
int main()
{
    double pi = 0.0, nthreads;
    double step = 1.0 / (double)num_steps;
    omp_set_num_threads( num_threads );
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x, sum;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id==0) nthreads = nthrds;
        for (i=id, sum=0.0; i<num_steps; i+=nthreads) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0 + x*x);
        }
        sum = sum * step;
        #pragma atomic
        pi += sum;
    }
    cout << "PI = " << pi << endl;
}
```

sum – z powrotem wartość skalarna osobno dla każdego wątku

synchronizacja zapewniona przez wydzielenie prostej części atomowej, wyniki podobne do wersji z poprawną sekcją krytyczną, np. dla 40 wątków obliczenie trwa 0.15 sekundy

SPMD *vs* podział pracy

- Obliczenie równoległe samo z siebie tworzy program typu ”Single Program Multiple Data”, tzn. każdy wątek niezależnie wykonuje **ten sam kod**
- **Worksharing** (podział pracy) – taka forma kodu, która pozwoli na rozdysponowanie części obliczeń pomiędzy wątki z tej samej grupy
  - pętla
  - sekcje
  - pojedynczy obiekt
  - zadania

Pętla – rozwiązanie SPMD

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

void work(long ww) {
    long sum = 0;
    for (long w = 0; w < ww; ++w) sum += w;
}

int main()
{
    const long max = 100, factor = 100000001;
    for (int i=0; i<max; ++i)  work((max - i) * factor);
}
```

Prosta pętla wykonująca mocno obciążającą funkcję – wykonanie sekwencyjne trwa około 2 min 35 sek.

Rozpisanie z podziałem sumy pomiędzy wątki, poprzez wyliczenie początku i końca (częściowej) pętli, dla każdego wątku osobno: uciążliwe. Wykonanie na 8 wątkach: około 35 sek.

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

void work(long ww) {
    long sum = 0;
    for (long w = 0; w < ww; ++w) sum += w;
}

int main()
{
    #pragma omp parallel num_threads(8)
    {
        const long max = 100, factor = 100000001;

        int istart, iend;
        int id = omp_get_thread_num();
        int nthrds = omp_get_num_threads();
        istart = id * max / nthrds;
        iend = ( id + 1 ) * max / nthrds;
        if ( id == nthrds - 1 ) iend = max;
        for (int i=istart; i<iend; ++i) work((max - i) * factor);
    }
}
```

Pętla - worksharing

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

void work(long ww) {
    long sum = 0;
    for (long w = 0; w < ww; ++w) sum += w;
}

int main()
{
    const long max = 100, factor = 100000001;

    #pragma omp parallel num_threads(8)
    #pragma omp for
        for (int i=0; i<max; ++i) work((max - i) * factor);
}
```

- Rozwiązanie OpenMP:
  #pragma omp for
  czas wykonania około 38 sek.
- Kompilator musi mieć wskazówkę jak rozdzielić pętlę między wątki!
- Sposób poinformowania kompilatora to klauzula schedule

Zapis bloku równoległego oraz zrównoleglonej pętli można połączyć:

```
double tab[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<MAX; ++i) {
        tab[i] = sth_big();
    }
}
```

```
double tab[MAX]; int i;
#pragma omp parallel for
{
    for (i=0; i<MAX; ++i) {
        tab[i] = sth_big();
    }
}
```

Praca z pętlami

- Znaleźć w programie pętle, podczas których wykonywane są **duże** obliczenia
- Zmodyfikować iteracje pętli do postaci niezależnej od kolejności ich wykonywania (nie zawsze możliwe)
- Umieścić właściwą dyrektywę OpenMP i przeprowadzić testy!

```
int i, j, tab[MAX];
j = 5;
for (i=0; i<MAX; ++i) {
    j += 2;
    tab[i] = big_calc(j);
}
```

Ta pętla nie może być zrównoleglona, gdyż jej kolejne iteracje zależne są od poprzedniej wartości (tutaj zmiennej j).

```
int i, tab[MAX];
#pragma omp parallel for
for (i=0; i<MAX; ++i) {
    int j = 5 + 2*(i+1);
    tab[i] = big_calc(j);
}
```

Stosowna modyfikacja, dzięki której zmienna j jest wyliczana na podstawie zmiennej i, zatem każda iteracja pętli może być wykonana niezależnie i można dokonać zrównoleglenia.

Pętla – klauzula schedule

Klauzula **schedule** określa w jaki sposób iteracje pętli są mapowane na wątki

- **schedule (static [,chunk])** rozdziel fragmenty pętli (o rozmiarze chunk) do każdego wątku
- **schedule (dynamic [,chunk])** każdy wątek zabiera część pętli (chunk) aż do momentu policzenia całej pętli
- **schedule (guided [,chunk])** wątki dynamicznie pobierają części pętli, pobrany rozmiar zaczyna się od dużego, by potem maleć do rozmiaru chunk, w miarę postępu obliczeń
- **schedule (auto)** decyzja o sposobie wykonania pozostawiona kompilatorowi, do czasu wykonania (może być inna niż powyższe)
- **schedule (runtime)** sposób wykonania (i rozmiar chunk) pobrane ze zmiennej OPM_SCHEDULE (lub poprzez procedury biblioteczne)

Pętla – którą klauzulę użyć

klauzula	użycie
static	obciążenie na iterację określone z góry i przewidziane przez programistę
dynamic	nieprzewidywalne, duże zmiany obciążenia pomiędzy iteracjami
guided	specjalna wersja klauzuli dynamic, redukująca dodatkowe narzuty
auto	gdy sposób wykonania może być wydedukowany z poprzednich wykonań tej samej pętli

decyzja podjęta na etapie kompilacji

decyzja podjęta podczas wykonania

Ustawienie trybu schedule(runtime) za pomocą zmiennej:

export OMP_SCHEDULE=*scheduling-type*

lub za pomocą funkcji bibliotecznej:

omp_set_schedule(*scheduling-type* **);**

Praca z pętlami zagnieżdżonymi

- Dla pętli zagnieżdżonych i niezależnych można zrównoleglić wielokrotne zagnieżdżone pętle za pomocą klauzuli collapse

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; ++i) {
    for (int j=0; j<M; ++j) {
        // ...
    }
}
```

- Uformowana zostanie pojedyncza pętla o rozmiarze N × M, a następnie zrównoleglona.

Sprawdźmy różnicę w działaniu z klauzulą collapse i bez niej!

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

int main() {
    #pragma omp parallel num_threads(18)
    {
        int id = omp_get_thread_num();
        #pragma omp for collapse(2)
        for (int i=0; i<5; ++i)
            for (int j=0; j<3; ++j)
                cout << "Watek nr " << id << " iteracja ("
                     << i << "," << j << ") " << endl;
    }
}
```

Pętle zależne - redukcja

- W wielu sytuacjach kolejne iteracje pętli są zależne od siebie.

```
double average=0.0; tab[MAX];
for (int i=0; i<MAX; ++i) {
    average += tab[i];
}
average /= MAX; // średnia
```

Liczenie średniej jest przykładem, w którym kolejny przebieg jest zależny od poprzedniego.

- Sytuacja taka ma swoją nazwę: redukcja (reduction)
- Wsparcie w OpenMP: klauzula **reduction(op : list)**
- Wewnątrz bloku równoległego lub podziału pracy
  - tworzona jest lokalna kopia zmiennej **list** zainicjalizowana w zależności od rodzaju **op** (np. 0 dla „+”)
  - aktualizowana jest lokalna kopia
  - lokalne kopie redukowane są do jednej wartości i połączone do jednej globalnej wartości

Pętle – redukcja, operandy

Zmienna ”list” musi być współdzielona:

```
double average=0.0; tab[MAX];
#pragma omp parallel for reduction (+:ave)
for (int i=0; i<MAX; ++i) {
    average += tab[i];
}
average /= MAX; // średnia
```

Wiele operatorów może być użytych w ramach mechanizmu redukcji. Wartości początkowe (tworzonych lokalnych zmiennych) zależą od rodzaju operatora (i są naturalne z matematycznego punktu widzenia) – patrz tabele:

operator	początkowa wartość
+	0
*	1
-	0
min	największa dodatnia wartość
max	najbardziej ujemna wartość

operator	początkowa wartość
&	~0
|	0
^	0
&&	1
||	0

Kod 4

g++ -fopenmp main.cc -o program

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

static long num_steps = 300000000;
double step;
int main()
{
    int i; double pi, sum = 0.0;
    step = 1.0 / (double)num_steps;
    #pragma omp parallel num_threads(8)
    {
        double x;
        #pragma omp for reduction(+:sum)
        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;
}
```

x – musi być zdefiniowana lokalnie, jako zmienna dla stosu każdego wątku

reduction(+:sum) rozłożenie zależnej pętli na obliczenia wielowątkowe

Wynik czasowy obliczeń:
0,14 sek, zatem porównywalny z poprzednimi technikami, a zarazem bardzo prosty i czytelny!

reduction działa na wszystkich wątkach (ich liczba ustawiona jak dyskutowano wcześniej)