Konstrukcje worksharing

Bariery (jawne, niejawne):

```
// fragment kodu
#pragma omp parallel shared(A, B, C) private(id)
{
    id = omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
        for(i=0; i<N; ++i) {
            C[i] = big_calc3(i,A);
        }
    #pragma omp for nowait
        for(i=0; i<N; ++i) {
            B[i] = big_calc2(C,i);
        }
    A[id] = big_calc4(id);
}
```

jawna bariera (wszystkie wątki czekają)

niejawna bariera (na końcu konstrukcji typu "worksharing"), można ją usunąć:

brak bariery (klauzula `nowait`)

niejawna bariera (na końcu bloku zrównoleglonego), nie można usunąć

Dyrektywy shared, private, default

Shared, private:

```
// fragment kodu
#pragma omp parallel shared(A,B,C) private(id)
{
    //...
}

#pragma omp parallel default(shared) private(i)
{
    //... default(shared) lub default(none)
}
```

shared: zmienne, wykorzystane w bloku, będą zmiennymi wspólnymi
private: zmienne, które będą prywatnymi (każdy wątek będzie miał swoją kopię)

Zmienne wspólne są dostępne dla każdego wątku, natomiast do danej zmiennej prywatnej ma dostęp tylko jeden określony wątek.

default(shared): wszystkie zmienne będą zmiennymi wspólnymi, jeśli jakieś mają być prywatne, to muszą być dalej wyspecyfikowane za pomocą private(…)
default(none): domyślnie zmienna nie jest ani współdzielona, ani prywatna, zatem taka dyrektywa wymusza jawną specyfikację shared(…) i private(…)
default(private): domyślnie zmienne będą prywatne, a jeśli mają być współdzielone, wymagają specyfikacji shared(…) default(private) MOŻLIWE TYLKO W FORTRANIE

firstprivate, lastprivate

```
// fragment kodu
int i = 10;
#pragma omp parallel firstprivate(i)
{
    //...
}
#pragma omp parallel for lastprivate(i)
// pętla for z jakimiś operacjami na i
// UWAGA: firstprivate() i lastprivate()
// mogą być razem użyte w pętli, ale nie
// na zmiennej roboczej tylko jakiejś innej

// wartość i utrwalona poza blokiem
```

```
#include <iostream>
#include <omp.h>
using namespace std;
int main () {
    int i = 10;
    #pragma omp parallel firstprivate(i) num_threads(8)
    {
        cout << "thread: " << omp_get_thread_num()
             << " i = " << i << endl;
        i = 1000 + omp_get_thread_num();
    }

    #pragma omp parallel
    {
        #pragma omp for lastprivate(i)
        for (i=0; i<50; ++i) {  }
    }
    cout << "the end: i = " << i << endl;
}
```

`firstprivate` i `lastprivate` to szczególne przypadki `private`

`firstprivate`: powoduje wprowadzenie wartości z kontekstu zewnętrznego do regionu równoległego

`lastprivate`: przenosi wartości z regionu równoległego do kontekstu zewnętrznego

Uzasadnieniem dla tych klas udostępniania danych jest to, że w regionie równoległym wszystkie zmienne prywatne zasłaniają te z kontekstu zewnętrznego, tj. nie jest możliwe użycie operacji przypisania do zmodyfikowania wartości zewnętrznej i z wnętrza regionu równoległego.

Przykład – jaka będzie wartość?

```
// mamy zmienne:
int A = 1, B = 1, C = 1;
// następuje blok:
#pragma omp parallel private(B) firstprivate(C)
```

- czy zmienne A, B, C są zmiennymi lokalnymi w każdym z wątków, czy też współdzielone w obszarze równoległym?
- jakie są ich początkowe wartości w wątkach w obszarze równoległym?

Wewnątrz obszaru #pragma omp parallel

- A jest zmienną współdzieloną i ma wartość (początkową) 1
- B i C są zmiennymi lokalnymi w każdym z wątków
  - początkowa wartość B jest niezdefiniowana
  - początkowa wartość C wynosi 1
- po opuszczeniu bloku równoległego lokalne kopie B i C przestają istnieć, wracamy do zmiennych B i C sprzed bloku, czyli posiadających wartość 1
- wartość końcowa A zależy od tego, czy została zmodyfikowana w bloku równoległym

Raz jeszcze obliczanie π - minimum zmian

```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
    int i;
    double x, pi, sum = 0.;
    const int N = 1000000000;
    const double w = 1.0/N;
    #pragma omp parallel for private(x) reduction(+:sum)
    for ( i = 0; i < N; ++i )
    {
        x = w*(i+0.5);
        sum = sum + 4.0/(1.0 + x*x);
    }

    pi = w*sum;
    cout << "Wynik PI = " << pi << endl;
}
```

Ten przykład ma jeszcze jedną wielką **zaletę**:
jeśli jakiś kompilator nie obsługiwałby biblioteki OpenMP, to linia z dyrektywą #pragma zostanie zignorowana, a sam program będzie działał dokładnie tak samo jak w wersji sekwencyjnej jednowątkowej.

Ta wersja różni się od wersji sekwencyjnej (pierwotnej) dodaniem tylko dwóch linii kodu więcej: nagłówka i dyrektywy #pragma
Zmienną x przekazano jako prywatną (każdy wątek ma swoją kopię).

Zmienna robocza **i** w pętli for jest domyślnie prywatna! Dotyczy to pierwszej (zewnętrznej) pętli zrównoleglonej. Gdyby np. była druga zagnieżdżona pętla, jej zmienna robocza musiałaby być też przekazana jako prywatna.

Konstrukcja z reduction(op:list) jest lepiej skalowalna niż rozwiązanie z sekcją krytyczną (critical).

Konstrukcja master

- konstrukcja `master` oznacza strukturalny blok, który zostanie wykonany tylko przez główny wątek (master thread, id 0)
- inne wątki przeskakują ten blok (nie ma żadnej synchronizacji)

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    {
        exchange_boundaries();
    }
    #pragma omp barrier
    do_many_other_things();
}
```

ponieważ nie ma synchronizacji na końcu bloku `master`, jeśli takiej potrzebujemy, konieczna jest jawna bariera

Konstrukcja single

- konstrukcja `single` oznacza strukturalny blok, który zostanie wykonany tylko przez jeden wątek (niekoniecznie główny), pierwszy, który zacznie wykonywanie tego bloku
- inne wątki muszą czekać na jego zakończenie (na końcu bloku jest niejawny punkt synchronizacji)

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
```

```
#pragma omp single nowait
{
    //...
}
```

możliwe jest zniesienie bariery na końcu bloku `single` (klauzula `nowait`)

na końcu nie trzeba pisać jawnie `barrier`, ponieważ jest tu niejawny punkt synchronizacji wątków (bariera)

Konstrukcja sections/section

- konstrukcja `sections/section` przekazuje osobne bloki do wykonania przez poszczególne wątki
- domyślnie na końcu bloku `sections` jest bariera, można ją wyłączyć za pomocą `nowait`

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            X_calculation();
        #pragma omp section
            y_calculation();
        #pragma omp section
            z_calculation();
    }
}
```

- fragmenty programu zawarte w różnych sekcjach muszą być od siebie niezależne, obliczenia wykonywane przez jeden wątek nie mogą jednocześnie odwoływać się do obliczeń wykonywanych przez wątek drugi (mogłoby to prowadzić do błędnego działania programu)
- każdy blok programu ograniczony nawiasami klamrowymi opcji section jest wykonywany przez przypisany mu wątek dokładnie raz, jeśli mamy do dyspozycji tylko jeden wątek obliczenia zawarte po dyrektywie sections zostaną przeprowadzone sekwencyjnie, nie mamy jednak wpływu na to w jakiej kolejności zostaną wywołane poszczególne bloki section

Synchronizacja niskopoziomowa: locks

Blokada (ang. `lock`) – mechanizm służący do zapobiegania konfliktom w dostępie do zasobów w środowiskach wielozadaniowych. Blokada wątku jest formą wzajemnego wykluczania (mutual exclusion). Blokada w OpenMP jest obiektem (omp_lock_t), który może być trzymany tylko przez jeden wątek w danej chwili.

Proste funkcje dostępne dla blokady:

- omp_init_lock(omp_lock_t*) – inicjalizacja blokady
- omp_set_lock(omp_lock_t*) – czeka aż blokada jest możliwa, po czym ją zakłada; żaden inny wątek nie może ustawić blokady aż nie zostanie zwolniona
- int omp_test_lock (omp_lock_t*) – próbuje założyć blokadę, ale nie blokuje wykonania wątku
- omp_unset_lock(omp_lock_t*) – zwolnienie blokady
- omp_destroy_lock(omp_lock_t*) – odwrotność omp_init_lock

Proste blokady – przykład

```
// fragment kodu
#pragma omp parallel for
    for(i=0; i<NBUCKETS; ++i) {
        omp_init_lock(&hist_locks[i]);
        hist[i] = 0;
    }
#pragma omp parallel for
    for(i=0; i<NVALS; ++i) {
        ival= (int)sample(arr[i]);
        omp_set_lock(&hist_locks[ival]);
            hist[ival]++;
        omp_unset_lock(&hist_locks[ival]);
    }
for(i=0; i<NBUCKETS; ++i)
    omp_destroy_lock(&hist_locks[i]);
```

jedna blokada na każdy element histogramu (tablicy)

wymuszenie wyłączności na modyfikację zawartości histogramu dla jednego wątku

zwolnienie zasobów

Blokady – uwagi

Blokada (ang. `lock`) – nie jest "tanim" mechanizmem synchronizacji. W podanym przykładzie zakłada się, że konflikty (próba modyfikacji tej samej komórki przez więcej niż jeden wątek w danej chwili) będą rzadkie, zakładanie blokady nie będzie więc zasadniczo spowalniać wykonania wątków.
Należy wykonać testy!

```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
    double pi = 0.,sum = 0., x;
    const int N = 10000000;
    const double w = 1.0/N;
    omp_lock_t writelock;
    omp_init_lock(&writelock);
    #pragma omp parallel private(x), firstprivate(sum), shared(pi)
    {
        #pragma omp for
        for (int i = 0; i < N; ++i)
        {
            x = w*(i-0.5);
            sum = sum + 4.0/(1.0 + x*x);
        }
        omp_set_lock(&writelock);
        pi = pi + w*sum;
        omp_unset_lock(&writelock);
    }
    omp_destroy_lock(&writelock);
    cout << "Wynik PI = " << pi << endl;
}
```

Runtime Library – procedury

- **Modyfikuj / sprawdź liczbę wątków**
  omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
- **Sprawdzenie czy jesteśmy w bloku zrównoleglonym?**
  omp_in_parallel()
- **Czy system ma dynamicznie dostosować liczbę wątków od jednego do drugiego bloku zrównoleglonego?**
  omp_set_dynamic(), omp_get_dynamic()
- **Ile procesorów ma system?**
  omp_get_num_procs()

Runtime Library – przykład

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

int main() {
  int num_threads;
  omp_set_dynamic(0);
  cout << "Liczba procesorow = " << omp_get_num_procs() << endl;
  omp_set_num_threads( omp_get_num_procs() );
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    #pragma omp single
    {
        num_threads = omp_get_num_threads();
        cout << "Watek nr " << id << " maks: " << num_threads << endl;
    }
  }
}
```

dynamiczne dostosowywanie liczby wątków zablokowane

zażądaj tyle wątków ile procesorów

ochrona operacji (wykonanie tylko przez jeden wątek)

System mimo wszystko może przydzielić mniej wątków, niż żądamy!

Zmienne środowiskowe

Zaleta stosowania zmiennych środowiskowych (environment variables) to możliwość konfigurowania sposobu wykonania programu bez konieczności rekompilacji.

OMP_NUM_THREADS int_val – ustawienie domyślnej wartości wątków do użycia, np. export OMP_NUM_THREADS=4

OMP_STACKSIZE – kontrolowanie wielkości stosu procesów potomnych, używane przyrostki B (bytes), K (Kilobytes), M (Megabytes), G (Gigabytes), lub T (Terabytes); domyślnie (jeśli nie podano jednostki) K (Kilobytes)

OMP_WAIT_POLICY – strategia zarządzania procesami w stanie bezczynności (idle); ACTIVE (wątki oczekujące na barierach / blokadach mają być aktywne), PASSIVE (wątki na barierach / blokadach mają zwalniać zasoby)