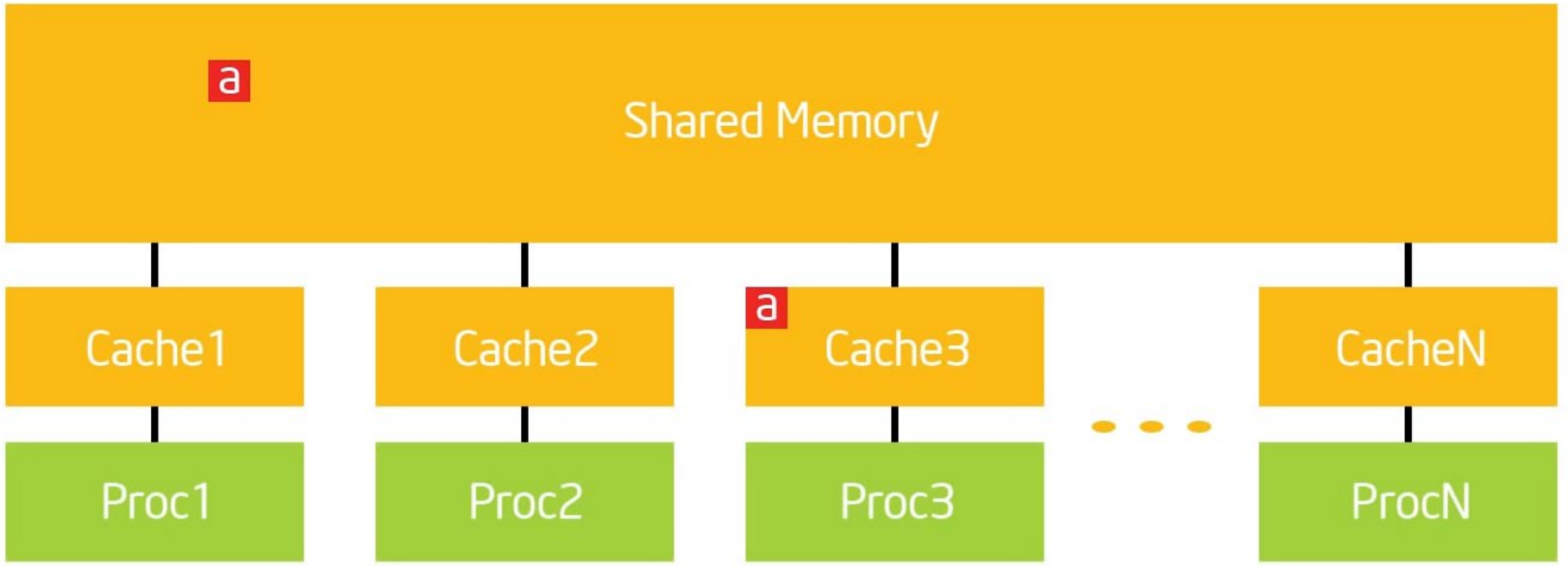
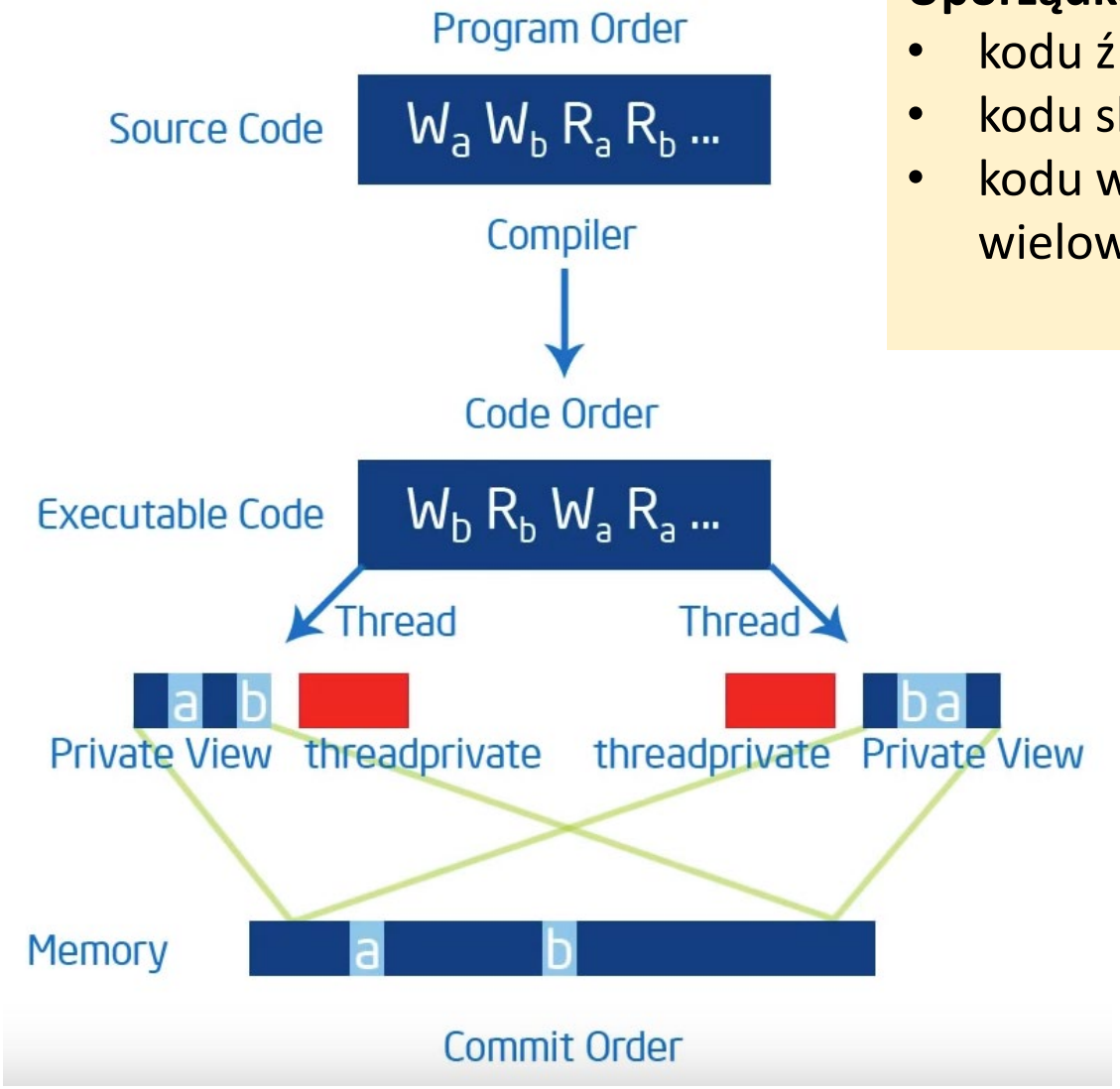
Model pamięci w OpenMP

OpenMP wspiera model pamięci współdzielonej.
Wszystkie wątki współdzielą pewną przestrzeń adresową, ale co w przypadku, gdy pewna zmienna jest w pamięci współdzielonej, oraz jest w pamięci podręcznej? Jaką zmienną np. odczyta Proc1?



Spójność operacji



Uporządkowanie na poziomie:

- kodu źródłowego
- kodu skompilowanego
- kodu wykonywanego w środowisku wielowątkowym / współdzielonym

Uporządkowanie: zmiana kolejności w dostępie do pamięci

- kompilator zmienia kolejność zapisaną w kodzie źródłowym na kolejność kodu skompilowanego
- komputer (system operacyjny) zmienia kolejność z kodu skompilowanego na kolejność w pamięci
- w danym momencie tymczasowy wygląd pamięci może się różnić od wyglądu pamięci współdzielonej

Modele spójności: oparte o uporządkowanie operacji typu odczyt Reads (R), zapis Writes (W), synchronizacja Synchronizations (S):
R->R, W->W, R->W, R->S, S->S, W->S

Spójność

Spójność sekwencyjna:

- w środowisku wieloprocesorowym operacje (R, W, S) są sekwencyjnie spójne, jeśli:
  - pozostają w porządku programu dla każdego procesora
  - są widziane w takim samym ogólnym porządku przez każdy z procesorów
  - porządek programu = porządek kodu = porządek wykonania
  - koszt: ogromny spadek wydajności

Poluzowana spójność:

- usunięcie części uwarunkowań (obostrzeń co do) spójności dla operacji R, W, S

OpenMP definiuje spójność jako wariant tzw. słabej spójności:

- operacje S muszą być w uporządkowanej kolejności pomiędzy wątkami
- nie można zmieniać kolejności operacji S z operacjami R lub W w tym samym wątku
  - "słaba" spójność gwarantuje: S->W, S->R, R->S, W->S, S->S

Niskopoziomowa synchronizacja: **flush**

Flush

- Definiuje punkt, w którym wątek ma zagwarantowany spójny widok pamięci w odniesieniu do "flush set".
- Flush set:
  - zbiór zmiennych widocznych w wątku dla bloku flush (użytym bez listy argumentów)
  - lista zmiennych w przypadku gdy użyta jest konstrukcja flush(list)
- Działanie flush gwarantuje, że:
  - wszystkie operacje R, W, przekrywające się z blokiem flush i mające miejsce przed flush, są wykonane wcześniej
  - wszystkie operacje R, W, przekrywające się z blokiem flush i mające miejsce po flush, są wykonane później
  - flush przekrywające się z innymi blokami flush nie mogą mieć zmienionej kolejności

Synchronizacja: flush

- flush gwarantuje konsystencję w tym, co widzą różne wątki

```
double A; // jakaś współdzielona zmienna
A = duze_obliczenie();
#pragma omp flush(A)
```

- w OpenMP flush zasadniczo jest używane najczęściej niejawnie:
  - przy wejściu / wyjściu do bloku zrównoleglonego
  - na jawnych / niejawnych barierach
  - przy wejściu / wyjściu do sekcji krytycznych
  - gdy blokada (zamek lock) jest ustawiana / zdejmowana (set / unset)
  - uwaga: używanie flush(list) ryzykowne gdy zmienne z list nie przekrywają się z flush, kompilator może zmienić kolejność wykonania

Producent - konsument

Klasyczny przykład programowania wielowątkowego: relacja producent-konsument, która wymaga synchronizacji pomiędzy dwoma wątkami.

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

void fill_rand(const int n, double* arr) {
        random_device rd;
        uniform_int_distribution<int> dist(1,20);
        for (int i = 0; i < n; ++i) {
            this_thread::sleep_for(0.02s);
            arr[i] = dist(rd);
        }
}

double sum_array( const int n, double* arr) {
    double s = 0;
    for (int i=0; i<n; ++i) s += arr[i];
    return s;
}
```

Punkt wyjściowy: kod sekwencyjny

```
int main() {

 const int N = 100;
 double sum = 0, runtime;
 int flag = 0;
 double* A = new double[N];

    runtime = omp_get_wtime();
    fill_rand(N, A);        // Producent: wypelnia tablice
    sum = sum_array(N, A);  // Konsument: liczy sume
    runtime = omp_get_wtime() - runtime;
    cout << "Po chwili: " << runtime << " suma = " << sum << endl;

 delete [] A;

}
```

Producent-konsument

```
int main() {

 const int N = 100;
 double sum = 0, runtime;
 int flag = 0;
 double*   A = new double[N];

    runtime = omp_get_wtime();
#pragma omp parallel sections num_threads(2)
{
 #pragma omp section
 {
    fill_rand(N, A);        // Producent
    #pragma omp flush
    flag = 1;
    #pragma omp flush(flag)
 }
 #pragma omp section
 {
    #pragma omp flush(flag)
    while (flag==0) {
        #pragma omp flush(flag)
    }
    #pragma omp flush
    sum = sum_array(N, A);  // Konsument
 }
}
    runtime = omp_get_wtime() -runtime;
    cout << "Po chwili: " << runtime << " suma = " << sum << endl;
}
```

W tym przykładzie jest zagrożenie **data race**

- w pierwszej sekcji flag przypisana jest wartość 1
- w drugiej sekcji czytana jest wartość flag

Rozwiązanie:

#pragma omp atomic [read|write|update|capture]

Producent-konsument

```
int main() {
 const int N = 100;
 double sum = 0, runtime;
 int flag = 0, flg_tmp;
 double* A = new double[N];

    runtime = omp_get_wtime();
#pragma omp parallel sections num_threads(2)
{
 #pragma omp section
 {
    fill_rand(N, A);        // Producent
    #pragma omp flush
    #pragma omp atomic write
    flag = 1;
    #pragma omp flush(flag)
 }
 #pragma omp section
 {
    while (true) {
        #pragma omp flush(flag)
        #pragma omp atomic read
        flg_tmp = flag;
        if (flag==1) break;
    }
    #pragma omp flush
    sum = sum_array(N, A);  // Konsument
 }
}
    runtime = omp_get_wtime() - runtime;
    cout << "Po chwili: " << runtime << " suma = " << sum << endl;
}
```

ta wersja jest całkowicie poprawna, operacje na zmiennej flag są zabezpieczone jako atomowe

w przykładzie, konsument rzeczywiście czeka na sygnał od producenta

jest to przykład na "współbieżne" (concurrent) programowanie

niemniej, sam przykład nie do końca odzwierciedla zysk z relacji producent-konsument, gdzie spodziewalibyśmy się raczej quasi-równoległej (parallel) relacji, podczas której konsument (prawie) od razu używa to, co wyprodukuje producent