Ministerstwo Nauki
i Szkolnictwa Wyższego

Regionalna Inicjatywa Doskonałości w Dyscyplinach
Informatyki, Elektrotechniki, Elektroniki, Automatyki i Robotyki
na Politechnice Częstochowskiej

Projekt w ramach programu Regionalna Inicjatywa Doskonałości, decyzja nr 020/RID/2018/19

Programowanie równoległe z wykorzystaniem współczesnych architektur komputerowych z pamięcią współdzieloną

Dr hab. inż. Łukasz Szustak
lszustak@icis.pcz.pl

Dr inż. Kamil Halbiniak
khalbniak@icis.pcz.pl

Politechnika Częstochowska

Programowanie równoległe z wykorzystaniem współczesnych architektur komputerowych z pamięcią współdzieloną

1. Wprowadzenie do współczesnych architektur komputerowych oraz programowania równoległego
2. Przegląd modeli i standardów programowania równoległego z pamięcią współdzieloną
3. Środowisko OpenMP programowania maszyn z pamięcią wspólna, w tym procesorów wielordzeniowych
4. Zrównoleglanie pętli oraz równoważenie obciążenia w systemach równoległych z pamięcią współdzieloną
5. Zrównoleglanie programów z wykorzystaniem modelu typu data parallelism
6. Zrównoleglanie programów z wykorzystaniem modelu typu task parallelism
7. Mapowanie obliczeń w hybrydowych systemach komputerowych z akceleratorami
8. Przykłady zrównoleglania obliczeń numerycznych

Programowanie równoległe z wykorzystaniem współczesnych architektur komputerowych z pamięcią współdzieloną

1. Wprowadzenie do współczesnych architektur komputerowych oraz programowania równoległego
2. Przegląd modeli i standardów programowania równoległego z pamięcią współdzieloną
3. Środowisko OpenMP programowania maszyn z pamięcią wspólna, w tym procesorów wielordzeniowych
4. Zrównoleglanie pętli oraz równoważenie obciążenia w systemach równoległych z pamięcią współdzieloną
5. Zrównoleglanie programów z wykorzystaniem modelu typu data parallelism
6. Zrównoleglanie programów z wykorzystaniem modelu typu task parallelism
7. **Mapowanie obliczeń w hybrydowych systemach komputerowych z akceleratorami**
8. Przykłady zrównoleglania obliczeń numerycznych

Ministerstwo Nauki
i Szkolnictwa Wyższego

Regionalna Inicjatywa Doskonałości w Dyscyplinach
Informatyki, Elektrotechniki, Elektroniki, Automatyki i Robotyki
na Politechnice Częstochowskiej

Projekt w ramach programu Regionalna Inicjatywa Doskonałości, decyzja nr 020/RID/2018/19

Temat nr 7

Mapowanie obliczeń w hybrydowych systemach komputerowych z akceleratorami

Dr inż. Kamil Halbiniak
khalbiniak@icis.pcz.pl

Politechnika Częstochowska

Agenda

- Wprowadzenie do platform hybrydowych
- Koncepcja modelu programowania architektur hybrydowych
- Model programowania akceleratorów bazujący na standardzie OpenMP

Hybrydowe platformy obliczeniowe

Wprowadzenie do platform hybrydowych

- Na przestrzeni ostatnich lat, jesteśmy świadkami rozwoju nowego trendu polegającego na stosowaniu dla coraz to szerszej gamy aplikacji heterogenicznych (lub hybrydowych) platform obliczeniowych
- Rozwiązania te bazują na integracji w różnych proporcjach dwóch głównych komponentów:
  - Tradycyjnych procesorów ogólnego przeznaczenia CPU
  - Wyspecjalizowanych masywnie równoległych akceleratorów obliczeniowych
- Hybrydowe platformy obliczeniowe w większości przypadków pozwalają na znaczne zwiększenie wydajności obliczeń
- Jednakże, w praktyce okazuje się, iż adaptacja aplikacji do platform heterogenicznych jest skomplikowanym procesem zarówno dla środowisk naukowych, jak i komercyjnych

Akceleratory obliczeń

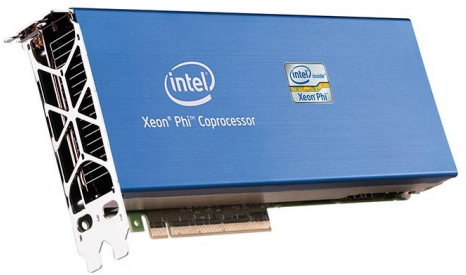
- Przykłady akceleratorów obliczeń:

GPU
(Graphics Processing Units)

Źródło: www.nvidia.com

Źródło: www.amd.com

Intel MIC
(Many Integrated Cores)



Źródło: www.intel.com

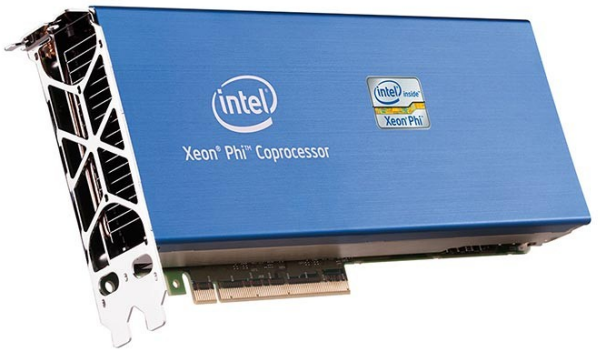
Akceleratory obliczeń

- Firma Intel zapowiedziała powstanie nowej architektury akceleratorów GPU: **Intel Xe**

Koncepcja akceleratora GPU firmy Intel. Źródło: www.wccftech.com

Akceleratory Intel MIC

- Architektura opracowana przez firmę Intel w 2010 roku
- Pierwsza generacja architektury – nazwa kodowa „Knights Corner”
  - Od 57 do 61 rdzeni (od 228 do 244 wątków) połączonych magistralą pierścieniową
  - Maksymalnie do 16 GB pamięci GDDR5
  - Akcelerator w formie koprocesora PCIe
  - ~1.2 Tflop/s dla obliczeń podwójnej precyzji
- Druga generacja architektury – nazwa kodowa „Knights Landing”
  - Od 64 do 72 rdzeni (od 256 do 288 wątków) tworzących siatkę 2D
  - 16 GB wbudowanej pamięci MCDRAM o wysokiej wydajności
  - Maksymalnie do 384 GB pamięci głównej
  - 3.4 Tflop/s dla obliczeń podwójnej precyzji uzyskanych m.in. dzięki przetwarzaniu ektorowemu SIMD danych 512-bitowych
  - Akcelerator w formie samodzielnego samodzielnego procesora przeznaczonego do obliczeń masywnie równoległych
  - Generacja ta jest kompatybilna binarnie z poprzednimi generacjami procesorów Intel



Hybrydowe platformy CPU-KNC

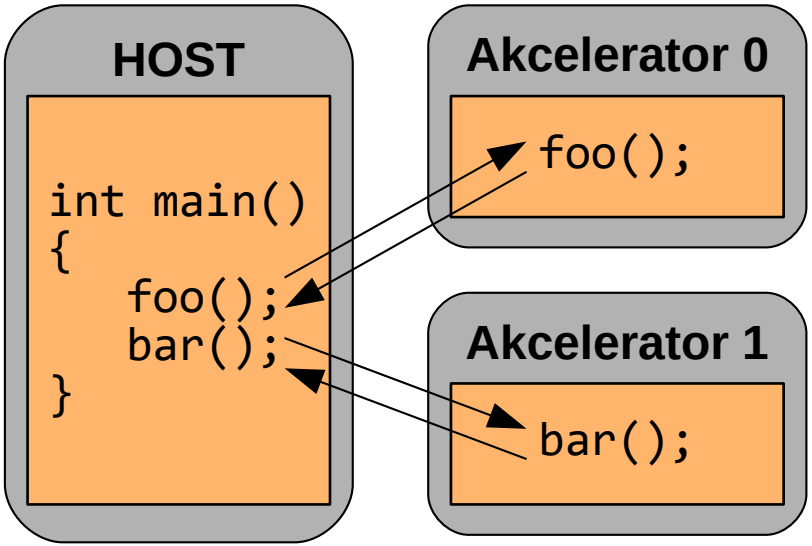
- Typowa platforma hybrydowa CPU-KNC składa się z:
  - Od 1 do 4 procesorów Intel Xeon
  - Od 1 do 8 akceleratorów Intel Xeon Phi
- Najczęściej stosowanym rozwiązaniem dla serwerów są węzły obliczeniowe wyposażone w dwa procesory ogólnego przeznaczenia oraz dwa układy KNC
- Platformy te pozwalają na uruchamianie aplikacji napisanych z wykorzystaniem popularnych standardów programowania równoległego (OpenMP, Intel Cilk Plus, Intel TBB, pThreads)
- Efektywne wykorzystanie dostępnych zasobów wymaga odmiennego odwzorowania algorytmów (w przypadku CPU oraz akceleratorów) oraz zapewnienia równoważenia obciążenia i minimalizacji kosztów komunikacji

Model programowania architektur hybrydowych CPU-KNC

- Osiągnięcie jak najwyższej wydajności obliczeń w architekturach hybrydowych wymaga jednoczesnego wykorzystania wszystkich dostępnych zasobów obliczeniowych
- Jednym ze sposobów osiągnięcia tego celu w przypadku platform wyposażonych w akceleratory KNC jest zastosowanie heterogenicznego modelu programowania bazującego na połączeniu obciążeniowego modelu programowania oraz modelu programowania równoległego z pamięcią współdzieloną
- W praktyce, pierwszy z nich pozwala na przesyłanie obliczeń do akceleratorów, podczas gdy drugi pozwala na wykorzystanie dostępnych jednostek obliczeniowych procesorów oraz akceleratorów

Odciążeniowy model programowania

- Odciążeniowy model programowania jest powszechnie stosowanym modelem programowania akceleratorów
- W modelu tym programista definiuje segmenty kodu źródłowego, które mają zostać zrealizowane w ramach akceleratora
- Program uruchamiany jest na procesorze hosta, a wybrane segmenty kodu są przesyłane do akceleratora w trakcie wykonania aplikacji



Programowanie akceleratorów w standardzie OpenMP: OpenMP Accelerator Model

OpenMP Accelerator Model

- Wraz z pojawieniem się czwartej wersji standardu OpenMP, model wykonawczy OpenMP został rozszerzony o mechanizm zwany OpenMP Accelerator Model
- Rozszerzenie to ma na celu uproszczenie procesu tworzenia wydajnego oprogramowania dedykowanego dla platform wyposażonych w masywnie równoległe akceleratory obliczeń
- OpenMP Accelerator Model zakłada, że platforma obliczeniowa składa się z wielu urządzeń podłączonych do hosta

Kopiowanie obliczeń i danych do akceleratorów – konstrukcja `target`

- Model wykonawczy OpenMP AM bazuje na modelu *host-centric*, w którym to host przesyła dane oraz obliczenia do akceleratorów
- Kopiowanie danych oraz obliczeń do akceleratorów możliwe jest z wykorzystaniem konstrukcji `target`:
  `#pragma omp target`
- Domyślnie, przesyłany do akceleratora blok kodu realizowany jest w sposób sekwencyjny do momentu napotkania odpowiedniej konstrukcji tworzącej nowe wątki
- OpenMP AM pozwala na określenie, który z akceleratorów w platformie wykorzystany zostanie do wykonania przesyłanego bloku kodu źródłowego za pomocą klauzuli `device(ID)`, gdzie ID to numer akceleratora
- Oprócz przesyłania danych do akceleratorów, OpenMP pozwala także na zrównoleglenie obliczeń realizowanych w ramach akceleratorów
- Konstrukcja `target` może zostać wywołana w ramach obszaru równoległego OpenMP oraz w ramach części programu wykonywanej przez wątek główny

Przykład I - OpenMP Accelerator Model

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            printf("Kod wykonywany przez CPU - liczba dostepnych watkow: %d\n", omp_get_num_threads());
        }
    }

    #pragma omp target
    {
        #pragma omp parallel
        {
            #pragma omp master
            {
                printf("Kod wykonywany przez MIC - liczba dostepnych watkow: %d\n", omp_get_num_threads());
            }
        }
    }

}
```

Kod wykonywany przez procesor CPU hosta

Kod wykonywany w ramach akceleratora

Przykład I - OpenMP Accelerator Model

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

int main()
{
    #pragma omp para
    {
        #pragma omp
        {
            printf("
        }
    }

    #pragma omp targ
    {
        #pragma omp
        {
            #pragma
            {
                prin
            }
        }
    }

}
```

```
khalbiniak@node01: ~/RID/T7
khalbiniak@node01:~/RID/T7$ icpc -O3 -fopenmp ./t7_omp1.cpp
khalbiniak@node01:~/RID/T7$ ./a.out
Kod wykonywany przez CPU - liczba dostepnych watkow: 72
Kod wykonywany przez MIC - liczba dostepnych watkow: 240
khalbiniak@node01:~/RID/T7$
```

Kompilujemy oraz uruchamiamy program

Wymiana danych pomiędzy hostem oraz akceleratorami

- Wykorzystanie akceleratorów w trybie obciążeniowym wymaga zazwyczaj wykonywanie operacji wymiany danych pomiędzy hostem oraz akceleratorami
- Wymiana danych w OpenMP Accelerator Model możliwe jest z wykorzystaniem klauzuli `map(attribute:list)` dyrektywy `#pragma omp target`
- Klauzula ta dostarcza cztery atrybuty pozwalające na określenie sposobu alokacji, inicjalizacji oraz kopiowania danych
  - Trzy z nich: `to`, `from` oraz `tofrom` pozwalają na niejawną alokację danych oraz określenie kierunku ich przesyłania
  - Atrybuty `to` oraz `from` realizują transfer danych odpowiednio do i z akceleratora, podczas gdy `tofrom` pozwala na przesyłanie danych w obu kierunkach
  - Ostatni atrybut – `alloc` – wykorzystywany jest jedynie wtedy, gdy wymagana jest jawna alokacja pamięci urządzenia bez potrzeby jej inicjalizacji

Zarządzanie pamięcią akceleratorów

- Wykorzystanie konstrukcji `target` do kopiowania danych oraz obliczeń domyślnie powoduje alokację oraz dealokację danych w pamięci akceleratora
- W przypadku programów, w których dyrektywa ta wywoływana jest wielokrotnie, zarządzanie pamięcią generuje zauważalne narzuty wydajnościowe
- W celu rozwiązania tego problemu OpenMP AM dostarcza dwie konstrukcje `target enter data` oraz `target exit data` pozwalające na zarządzanie pamięcią akceleratora
  - Pierwsza z nich wykorzystywana jest do alokacji danych w pamięci akceleratorów
  - Druga z nich zwalnia uprzednio przydzieloną pamięć
- Konstrukcje te wykorzystywane są w połączeniu z klauzulą `map`
- Rozwiązanie to pozwala na zapewnienie trwałości danych pomiędzy obliczeniami przesyłanymi wielokrotnie do akceleratora

Synchronizacja buforów hosta oraz akceleratorów – konstrukcja `target udpate`

- Konstrukcja `target update` pozwala na synchronizację buforów hosta oraz akceleratorów
- Dyrektywa ta ma jednak jedynie zastosowanie w ramach obszaru danych utworzonego za pomocą konstrukcji `enter data` oraz `exit data`
- Kierunek przesyłania danych określany jest z wykorzystaniem klauzul `to(list)` oraz `from(list)`
- Konstrukcja `target update` pozwala również na przesyłanie danych w trakcie obliczeń realizowanych przez akcelerator (asynchroniczny transfer danych z punktu widzenia akceleratora)
- Konstrukcja ta podobnie do konstrukcji `target`, może zostać wywołana w ramach obszaru równoległego oraz w ramach części programu realizowanej przez wątek główny

Deklarowanie fragmentów programu mapowanych do akcleratora

- OMPAM dostarcza konstrukcję `declare target` do określenia fragmentów kodu źródłowego oraz zmiennych mapowanych do akceleratora
- Pozwala to kompilatorowi na wygenerowanie kodu binarnego, który następnie może zostać wywołany wewnątrz bloku przesyłanego akcelerator z wykorzystaniem dyrektywy `#pragma omp target`
- Blok kodu, który ma zostać skompilowany dla akceleratora należy umieścić pomiędzy dwiema dyrektywami `#pragma omp declare target` oraz `#pragma omp end declare target`

Tryb asynchroniczny

- Domyślnie w OpenMP Accelerator Model, wątek wywołujący konstrukcje `target` oraz `target update` wstrzymywany jest do czasu ich wykonania
- OpenMP Accelerator Model pozwala na asynchroniczną realizację obu tych dyrektyw (asynchroniczną z punktu widzenia hosta)
- Osiągnięcie tego celu możliwe jest poprzez zastosowanie dedykowanej klauzuli `nowait`
- Użycie klauzuli `nowait` w połączeniu z dyrektywą `target` gwarantuje, że przesyłany do akceleratora blok kodu wykonywany jest w tle, a wątek natychmiastowo kontynuuje realizację programu
- Podobna sytuacja ma miejsce w przypadku konstrukcji `target update`, wówczas synchronizacja buforów hosta oraz akceleratora wykonywana jest w sposób asynchronicznie

Definiowanie kolejności realizacji zadań w trybie asynchronicznym

- OpenMP Accelerator Model pozwala określić kolejność wykonania konstrukcji `target` oraz `target update` w trybie asynchronicznym
- W tym celu należy wykorzystać dedykowaną klauzulę: `depend(rodzaj_zaleznosci:zmienne)`
  - Klauzula ta wykorzystywana jest również do określenia kolejności wykonywania zadań OpenMP
  - Trzy rodzaje zależności: `in`, `out`, `inout`

Synchronizacja urządzeń w trybie asynchronicznym

- Synchronizacja hosta oraz akceleratorów możliwa jest zarówno w ramach obszaru równoległego OpenMP jak i części programu realizowanej przez wątek główny
- Osiągnięcie tego celu możliwe jest poprzez użycie następujących dyrektyw:
  - `#pragma omp taskwait` (dyrektywa ta może zostać wykorzystana do synchronizacji hosta oraz akceleratorów w ramach obszaru równoległego OpenMP oraz poza nim)
  - `#pragma omp barrier` (dyrektywa ta może zostać wykorzystana do synchronizacji urządzeń w ramach obszaru równoległego OpenMP)
- Synchronizacja hosta oraz akceleratorów w obszarze równoległym realizowana również w ramach niejawnej synchronizacji, występującej przykładowo na końcu bloku poprzedzonego dyrektywą `#pragma omp for`

Przykład II - OpenMP Accelerator Model

```
#pragma omp target enter data map(to: B[0:n], C[0:n]) map(alloc: A[0:n])

int x = 0;

for(int t=0; t<num_steps; ++t)
{
    #pragma omp target map(to: A[0:n], B[0:n], C[0:n]) nowait depend(out: x)
    {
        mic_computations(A, B, C, n);
    }
    #pragma omp target update from(A[0:n]) nowait depend(inout: x)

    host_activity();

    #pragma omp taskwait

}

#pragma omp target exit data  map(from: A[0:n],  B[0:n], C[0:n])
```

Przykład II - OpenMP Accelerator Model

Alokujemy dane w pamięci akceleratorów

```
#pragma omp target enter data map(to: B[0:n], C[0:n]) map(alloc: A[0:n])

int x = 0;

for(int t=0; t<num_steps; ++t)
{
    #pragma omp target map(to: A[0:n], B[0:n], C[0:n]) nowait depend(out: x)
    {
        mic_computations(A, B, C, n);
    }
    #pragma omp target update from(A[0:n]) nowait depend(inout: x)

    host_activity();

    #pragma omp taskwait

}

#pragma omp target exit data  map(from: A[0:n],  B[0:n], C[0:n])
```

Przykład II - OpenMP Accelerator Model

Alokujemy dane w pamięci akceleratorów

W zaprezentowanym kodzie operacje inicjalizacji tablicy `A` oraz transferu tablic `B` i `C` realizowane są z wykorzystaniem klauzuli `map`

```
#pragma omp target enter data map(to: B[0:n], C[0:n]) map(alloc: A[0:n])

int x = 0;

for(int t=0; t<num_steps; ++t)
{
    #pragma omp target map(to: A[0:n], B[0:n], C[0:n]) nowait depend(out: x)
    {
        mic_computations(A, B, C, n);
    }
    #pragma omp target update from(A[0:n]) nowait depend(inout: x)

    host_activity();

    #pragma omp taskwait

}

#pragma omp target exit data  map(from: A[0:n],  B[0:n], C[0:n])
```

Przykład II - OpenMP Accelerator Model

Alokujemy dane w pamięci akceleratorów

W zaprezentowanym kodzie operacje inicjalizacji tablicy `A` oraz transferu tablic `B` i `C` realizowane są z wykorzystaniem klauzuli `map`

Obliczenia w ramach akceleratora realizowane są asynchronicznie

```
#pragma omp target enter data map(to: B[0:n], C[0:n]) map(alloc: A[0:n])

int x = 0;

for(int t=0; t<num_steps; ++t)
{
    #pragma omp target map(to: A[0:n], B[0:n], C[0:n]) nowait depend(out: x)
    {
        mic_computations(A, B, C, n);
    }
    #pragma omp target update from(A[0:n]) nowait depend(inout: x)

    host_activity();

    #pragma omp taskwait

}

#pragma omp target exit data  map(from: A[0:n],  B[0:n], C[0:n])
```

Przykład II - OpenMP Accelerator Model

Alokujemy dane w pamięci akceleratorów

W zaprezentowanym kodzie operacje inicjalizacji tablicy `A` oraz transferu tablic `B` i `C` realizowane są z wykorzystaniem klauzuli `map`

```
#pragma omp target enter data map(to: B[0:n], C[0:n]) map(alloc: A[0:n])

int x = 0;

for(int t=0; t<num_steps; ++t)
{
    #pragma omp target map(to: A[0:n], B[0:n], C[0:n]) nowait depend(out: x)
    {
        mic_computations(A, B, C, n);
    }
    #pragma omp target update from(A[0:n]) nowait depend(inout: x)

    host_activity();

    #pragma omp taskwait

}

#pragma omp target exit data  map(from: A[0:n],  B[0:n], C[0:n])
```

Obliczenia w ramach akceleratora realizowane są asynchronicznie

Wyniki obliczeń transferowane są do hosta w tle

Przykład II - OpenMP Accelerator Model

Alokujemy dane w pamięci akceleratorów

W zaprezentowanym kodzie operacje inicjalizacji tablicy `A` oraz transferu tablic `B` i `C` realizowane są z wykorzystaniem klauzuli `map`

```
#pragma omp target enter data map(to: B[0:n], C[0:n]) map(alloc: A[0:n])

int x = 0;

for(int t=0; t<num_steps; ++t)
{
    #pragma omp target map(to: A[0:n], B[0:n], C[0:n]) nowait depend(out: x)
    {
        mic_computations(A, B, C, n);
    }
    #pragma omp target update from(A[0:n]) nowait depend(inout: x)

    host_activity();

    #pragma omp taskwait

}

#pragma omp target exit data  map(from: A[0:n],  B[0:n], C[0:n])
```

Obliczenia w ramach akceleratora realizowane są asynchronicznie

klauzula depend w konstrukcjach `target` oraz `target update` definiuje kolejność ich wykonania w trybie asynchronicznym

Wyniki obliczeń transferowane są do hosta w tle

Przykład II - OpenMP Accelerator Model

Alokujemy dane w pamięci akceleratorów

W zaprezentowanym kodzie operacje inicjalizacji tablicy `A` oraz transferu tablic `B` i `C` realizowane są z wykorzystaniem klauzuli `map`

```
#pragma omp target enter data map(to: B[0:n], C[0:n]) map(alloc: A[0:n])

int x = 0;

for(int t=0; t<num_steps; ++t)
{
    #pragma omp target map(to: A[0:n], B[0:n], C[0:n]) nowait depend(out: x)
    {
        mic_computations(A, B, C, n);
    }
    #pragma omp target update from(A[0:n]) nowait depend(inout: x)

    host_activity();

    #pragma omp taskwait

}

#pragma omp target exit data  map(from: A[0:n],  B[0:n], C[0:n])
```

Obliczenia w ramach akceleratora realizowane są asynchronicznie

klauzula `depend` w konstrukcjach `target` oraz `target update` definiuje kolejność ich wykonania w trybie asynchronicznym

Wyniki obliczeń transferowane są do hosta w tle

Host wykonuje zadanie, w trakcie kiedy akcelerator wykonuje obliczenia

Przykład II - OpenMP Accelerator Model

Alokujemy dane w pamięci akceleratorów

W zaprezentowanym kodzie operacje inicjalizacji tablicy `A` oraz transferu tablic `B` i `C` realizowane są z wykorzystaniem klauzuli `map`

```
#pragma omp target enter data map(to: B[0:n], C[0:n]) map(alloc: A[0:n])

int x = 0;

for(int t=0; t<num_steps; ++t)
{
    #pragma omp target map(to: A[0:n], B[0:n], C[0:n]) nowait depend(out: x)
    {
        mic_computations(A, B, C, n);
    }
    #pragma omp target update from(A[0:n]) nowait depend(inout: x)

    host_activity();

    #pragma omp taskwait
}

#pragma omp target exit data  map(from: A[0:n],  B[0:n], C[0:n])
```

Obliczenia w ramach akceleratora realizowane są asynchronicznie

klauzula `depend` w konstrukcjach `target` oraz `target update` definiuje kolejność ich wykonania w trybie asynchronicznym

Wyniki obliczeń transferowane są do hosta w tle

Host wykonuje zadanie, w trakcie kiedy akcelerator wykonuje obliczenia

Synchronizacja hosta oraz akceleratora

Przykład II - OpenMP Accelerator Model

Alokujemy dane w pamięci akceleratorów

W zaprezentowanym kodzie operacje inicjalizacji tablicy `A` oraz transferu tablic `B` i `C` realizowane są z wykorzystaniem klauzuli `map`

```
#pragma omp target enter data map(to: B[0:n], C[0:n]) map(alloc: A[0:n])

int x = 0;

for(int t=0; t<num_steps; ++t)
{
    #pragma omp target map(to: A[0:n], B[0:n], C[0:n]) nowait depend(out: x)
    {
        mic_computations(A, B, C, n);
    }
    #pragma omp target update from(A[0:n]) nowait depend(inout: x)

    host_activity();

    #pragma omp taskwait
}

#pragma omp target exit data  map(from: A[0:n],  B[0:n], C[0:n])
```

Obliczenia w ramach akceleratora realizowane są asynchronicznie

klauzula `depend` w konstrukcjach `target` oraz `target update` definiuje kolejność ich wykonania w trybie asynchronicznym

Wyniki obliczeń transferowane są do hosta w tle

Host wykonuje zadanie, w trakcie kiedy akcelerator wykonuje obliczenia

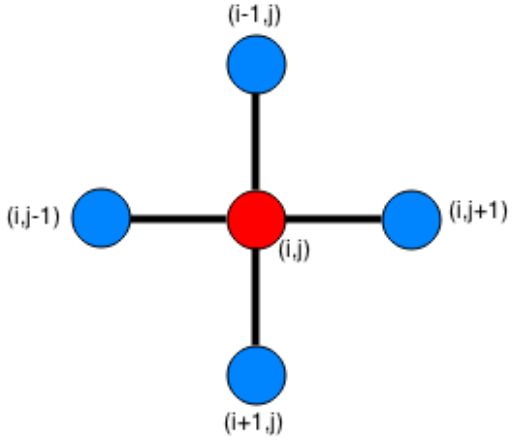
Synchronizacja hosta oraz akceleratora

Zwalniamy pamięć akceleratora

Aplikacja: obliczenia typu stencil

Algorytmy typu stencil

- Grupa obliczeń stosowana w szerokiej gamie aplikacji
- Wartość danego elementu wyznaczana jest na podstawie wartości elementów sąsiednich
- Obliczenia typu *stencil* na ogół nie osiągają maksymalnej wydajności jaką oferuje architektura




```
for(int ts=0; ts<timeSteps; ++ts)
{
    for(int i=1; i<n-1; ++i)
    {
        for(int j=1; j<n-1; ++j)
        {
            B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
        }
    }

    for(int i=1; i<n-1; ++i)
    {
        for(int j=1; j<n-1; ++j)
        {
            A[i*n+j] = B[i*n+j];
        }
    }
}
```

Algorytm: wersja 2 x CPU

```
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
```

```
int main()
{
    cout << "==========  CPU only ==========" << endl;
    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i=0; i<n; ++i)
    {
        for(int j=0; j<n; ++j)
        {
            A1[i*n+j] = 1 + (double)rand() / RAND_MAX;
            A2[i*n+j] = A1[i*n+j];
            B1[i*n+j] = 0.0;
            B2[i*n+j] = 0.0;
        }
    }

    //...

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        stencil_code_parallel(A2, B2, n);
    }
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //...

    return 0;
}
```

Algorytm: wersja 2 x CPU

```
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
```

Zrównoleglamy obliczenia
z wykorzystaniem OpenMP

```
int main()
{
    cout << "==========  CPU only ==========" << endl;
    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i=0; i<n; ++i)
    {
        for(int j=0; j<n; ++j)
        {
            A1[i*n+j] = 1 + (double)rand() / RAND_MAX;
            A2[i*n+j] = A1[i*n+j];
            B1[i*n+j] = 0.0;
            B2[i*n+j] = 0.0;
        }
    }

    //...

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        stencil_code_parallel(A2, B2, n);
    }
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //...

    return 0;
}
```

Algorytm: wersja 2 x CPU

```
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
```

Zrównoleglamy obliczenia z wykorzystaniem OpenMP

Dane wyjściowe z jednego kroku czasowego są danymi wejściowymi w kroku kolejnym

```
int main()
{
    cout << "==========  CPU only ==========" << endl;
    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i=0; i<n; ++i)
    {
        for(int j=0; j<n; ++j)
        {
            A1[i*n+j] = 1 + (double)rand() / RAND_MAX;
            A2[i*n+j] = A1[i*n+j];
            B1[i*n+j] = 0.0;
            B2[i*n+j] = 0.0;
        }
    }

    //...

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        stencil_code_parallel(A2, B2, n);
    }
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //...

    return 0;
}
```

Algorytm: wersja 2 x CPU

```
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
```

Zrównoleglamy obliczenia z wykorzystaniem OpenMP

Tworzymy tablice oraz wypełniamy je w sposób równoległy

Dane wyjściowe z jednego kroku czasowego są danymi wejściowymi w kroku kolejnym

```
int main()
{
    cout << "==========  CPU only ==========" << endl;
    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i=0; i<n; ++i)
    {
        for(int j=0; j<n; ++j)
        {
            A1[i*n+j] = 1 + (double)rand() / RAND_MAX;
            A2[i*n+j] = A1[i*n+j];
            B1[i*n+j] = 0.0;
            B2[i*n+j] = 0.0;
        }
    }

    //...

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        stencil_code_parallel(A2, B2, n);
    }
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //...

    return 0;
}
```

Algorytm: wersja 2 x CPU

```
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
```

Zrównoleglamy obliczenia z wykorzystaniem OpenMP

Tworzymy tablice oraz wypełniamy je w sposób równoległy

Polityka pamięci *first-touch*. Dane znajdują się w pamięci procesora, który pierwszy je „dotknął”.

Dane wyjściowe z jednego kroku czasowego są danymi wejściowymi w kroku kolejnym

```
int main()
{
    cout << "==========  CPU only ==========" << endl;
    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i=0; i<n; ++i)
    {
        for(int j=0; j<n; ++j)
        {
            A1[i*n+j] = 1 + (double)rand() / RAND_MAX;
            A2[i*n+j] = A1[i*n+j];
            B1[i*n+j] = 0.0;
            B2[i*n+j] = 0.0;
        }
    }

    //...

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        stencil_code_parallel(A2, B2, n);
    }
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //...

    return 0;
}
```

Algorytm: wersja 2 x CPU

```
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
```

Zrównoleglamy obliczenia z wykorzystaniem OpenMP

Dane wyjściowe z jednego kroku czasowego są danymi wejściowymi w kroku kolejnym

```
int main()
{
    cout << "==========  CPU only ==========" << endl;
    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i=0; i<n; ++i)
    {
        for(int j=0; j<n; ++j)
        {
            A1[i*n+j] = 1 + (double)rand() / RAND_MAX;
            A2[i*n+j] = A1[i*n+j];
            B1[i*n+j] = 0.0;
            B2[i*n+j] = 0.0;
        }
    }

    //...

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        stencil_code_parallel(A2, B2, n);
    }
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //...

    return 0;
}
```

Tworzymy tablice oraz wypełniamy je w sposób równoległy

Polityka pamięci *first-touch*. Dane znajdują się w pamięci procesora, który pierwszy je „dotknął”.

Uruchamiamy obliczenia z wykorzystaniem dwóch procesorów CPU oraz mierzymy czas wykonania

Algorytm: wersja 2 x CPU

```
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
```

Zrównoleglamy obliczenia z wykorzystaniem OpenMP

Dane wyjściowe z jednego kroku czasowego są danymi wejściowymi w kroku kolejnym

Przepisywanie danych w drugiej pętli zrealizować możemy za pomocą zamiany wskaźników

```
int main()
{
    cout << "==========  CPU only ==========" << endl;
    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i=0; i<n; ++i)
    {
        for(int j=0; j<n; ++j)
        {
            A1[i*n+j] = 1 + (double)rand() / RAND_MAX;
            A2[i*n+j] = A1[i*n+j];
            B1[i*n+j] = 0.0;
            B2[i*n+j] = 0.0;
        }
    }

    //...

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        stencil_code_parallel(A2, B2, n);
    }
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //...

    return 0;
}
```

Tworzymy tablice oraz wypełniamy je w sposób równoległy

Polityka pamięci *first-touch*. Dane znajdują się w pamięci procesora, który pierwszy je „dotknął”.

Uruchamiamy obliczenia z wykorzystaniem dwóch procesorów CPU oraz mierzymy czas wykonania

Algorytm: wersja 2 x CPU

```
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp f
        for(int i=1;
        {
            for(int j
            {
                A[i*n
            }
        }
    }
}
```

```
int main()
{
    cout << "==========  CPU only ==========" << endl;
    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];

                                                    ads(NUM_THREADS)

                                                    and() / RAND_MAX;

                                                    "\n";
    start  = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        stencil_code_parallel(A2, B2, n);
    }
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //...

    return 0;
}
```

Zrównoleglamy obliczenia z wykorzystaniem OpenMP

Zasoby obliczeniowe	Czas obliczeń [s]	Przyspieszenie
2 x CPU	8.58	---

Kroki czasowe: 200, rozmiar macierzy: 10000x10000
Kompilator: icpc 17.0.1, flagi: -O3 -fopenmp
Platforma: 2 x Intel Xeon E5-2699 v3 + 2 x Intel Xeon Phi 7120P

Dane wy
kroku cza
wejściowymi w kroku kolejnym

pierwszy je „dotknął”.

Przepisywanie danych w drugiej pętli zrealizować możemy za pomocą zamiany wskaźników

Uruchamiamy obliczenia z wykorzystaniem dwóch procesorów CPU oraz mierzymy czas wykonania

Algorytm: wersja 1 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

```
int main()
{
    cout << "==========  Intel MIC ==========" << endl;

    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    // ...

    #pragma omp target enter data map(to: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    #pragma omp target map(to: A2[0:n*n], B2[0:n*n])
    {
        for(int ts=0; ts<timeSteps; ++ts)
        {
            stencil_code_parallel(A2, B2, n);
        }
    }
    #pragma omp target update from(B2[0:n*n])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    #pragma omp target exit data map(from: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    //...

    delete[] B2;
    delete[] B1;
    delete[] A2;
    delete[] A1;
    return 0;
}
```

Algorytm: wersja 1 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Kod realizowany w ramach akceleratora

```
int main()
{
    cout << "==========  Intel MIC ==========" << endl;

    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    // ...

    #pragma omp target enter data map(to: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    #pragma omp target map(to: A2[0:n*n], B2[0:n*n])
    {
        for(int ts=0; ts<timeSteps; ++ts)
        {
            stencil_code_parallel(A2, B2, n);
        }
    }
    #pragma omp target update from(B2[0:n*n])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    #pragma omp target exit data map(from: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    //...

    delete[] B2;
    delete[] B1;
    delete[] A2;
    delete[] A1;
    return 0;
}
```

Algorytm: wersja 1 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Kod realizowany w ramach akceleratora

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    cout << "==========  Intel MIC ==========" << endl;

    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    // ...

    #pragma omp target enter data map(to: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    #pragma omp target map(to: A2[0:n*n], B2[0:n*n])
    {
        for(int ts=0; ts<timeSteps; ++ts)
        {
            stencil_code_parallel(A2, B2, n);
        }
    }
    #pragma omp target update from(B2[0:n*n])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    #pragma omp target exit data map(from: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    //...

    delete[] B2;
    delete[] B1;
    delete[] A2;
    delete[] A1;
    return 0;
}
```

Algorytm: wersja 1 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Kod realizowany w ramach akceleratora

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    cout << "==========  Intel MIC ==========" << endl;

    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    // ...

    #pragma omp target enter data map(to: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    #pragma omp target map(to: A2[0:n*n], B2[0:n*n])
    {
        for(int ts=0; ts<timeSteps; ++ts)
        {
            stencil_code_parallel(A2, B2, n);
        }
    }
    #pragma omp target update from(B2[0:n*n])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    #pragma omp target exit data map(from: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    //...

    delete[] B2;
    delete[] B1;
    delete[] A2;
    delete[] A1;
    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Algorytm: wersja 1 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Kod realizowany w ramach akceleratora

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    cout << "==========  Intel MIC ==========" << endl;

    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    // ...

    #pragma omp target enter data map(to: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    #pragma omp target map(to: A2[0:n*n], B2[0:n*n])
    {
        for(int ts=0; ts<timeSteps; ++ts)
        {
            stencil_code_parallel(A2, B2, n);
        }
    }
    #pragma omp target update from(B2[0:n*n])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    #pragma omp target exit data map(from: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    //...

    delete[] B2;
    delete[] B1;
    delete[] A2;
    delete[] A1;
    return 0;
}
```

Alokujemy dane w pamięci akceleratora

Tworzymy oraz wypełniamy tablicę

Algorytm: wersja 1 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Kod realizowany w ramach akceleratora

Alokujemy dane w pamięci akceleratora

Uruchamiamy obliczenia dla zdefiniowanej liczby kroków czasowych oraz mierzymy czas

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    cout << "==========  Intel MIC ==========" << endl;

    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    // ...

    #pragma omp target enter data map(to: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    #pragma omp target map(to: A2[0:n*n], B2[0:n*n])
    {
        for(int ts=0; ts<timeSteps; ++ts)
        {
            stencil_code_parallel(A2, B2, n);
        }
    }
    #pragma omp target update from(B2[0:n*n])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    #pragma omp target exit data map(from: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    //...

    delete[] B2;
    delete[] B1;
    delete[] A2;
    delete[] A1;
    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Algorytm: wersja 1 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Kod realizowany w ramach akceleratora

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    cout << "==========  Intel MIC ==========" << endl;

    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    // ...

    #pragma omp target enter data map(to: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    #pragma omp target map(to: A2[0:n*n], B2[0:n*n])
    {
        for(int ts=0; ts<timeSteps; ++ts)
        {
            stencil_code_parallel(A2, B2, n);
        }
    }
    #pragma omp target update from(B2[0:n*n])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    #pragma omp target exit data map(from: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    //...

    delete[] B2;
    delete[] B1;
    delete[] A2;
    delete[] A1;
    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Alokujemy dane w pamięci akceleratora

Uruchamiamy obliczenia dla zdefiniowanej liczby kroków czasowych oraz mierzymy czas

Po zakończeniu obliczeń przesyłamy dane do hosta

Algorytm: wersja 1 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Kod realizowany w ramach akceleratora

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    cout << "==========  Intel MIC ==========" << endl;

    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
    double *B1 = new double[n*n];
    double *A2 = new double[n*n];
    double *B2 = new double[n*n];

    // ...

    #pragma omp target enter data map(to: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    cout << "Wersja rownolegla..." << "\n";
    start  = omp_get_wtime();
    #pragma omp target map(to: A2[0:n*n], B2[0:n*n])
    {
        for(int ts=0; ts<timeSteps; ++ts)
        {
            stencil_code_parallel(A2, B2, n);
        }
    }
    #pragma omp target update from(B2[0:n*n])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    #pragma omp target exit data map(from: A1[0:n*n], A2[0:n*n], B1[0:n*n], B2[0:n*n])

    //...

    delete[] B2;
    delete[] B1;
    delete[] A2;
    delete[] A1;
    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Alokujemy dane w pamięci akceleratora

Uruchamiamy obliczenia dla zdefiniowanej liczby kroków czasowych oraz mierzymy czas

Po zakończeniu obliczeń przesyłamy dane do hosta

Zwalniamy pamięć koprocesora

Algorytm: wersja 1 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=1; i<n-1; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=1; i<
        {
            for(int j=1
            {
                A[i*n+j
            }
        }
    }
}
#pragma omp end declare
```

Kod realizowany w ramach akceleratora

```
int main()
{
    cout << "==========  Intel MIC ==========" << endl;

    srand(time(0));

    const int timeSteps = 200;
    double start, stop;

    const int n = 10000;
    double *A1 = new double[n*n];
```

Tworzymy oraz wypełniamy tablicę

Zasoby obliczeniowe	Czas obliczeń [s]	Przyspieszenie
2 x CPU	8,58	---
1 x Intel MIC	6,53	1,31x

Kroki czasowe: 200, rozmiar macierzy: 10000x10000
Kompilator: icpc 17.0.1, flagi: -O3 -fopenmp
Platforma: 2 x Intel Xeon E5-2699 v3 + 2 x Intel Xeon Phi 7120P

Obliczenia r
W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

przesyłamy dane do hosta

```
    //...

    delete[] B2;
    delete[] B1;
    delete[] A2;
    delete[] A1;
    return 0;
}
```

Zwalniamy pamięć koprocesora

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
    int mic1_halo_out = rowsPerDevice * n;
    int m0, m1;

    start = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        #pragma omp target update device(0) to(A2[mic0_halo_in : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) to(A2[mic1_halo_in : n]) nowait depend(inout:m1)

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(0) nowait depend(inout:m0)
        {
            stencil_code_parallel(A2, B2, 1, rowsPerDevice, n);
        }

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(1) nowait depend(inout:m1)
        {
            stencil_code_parallel(A2, B2, rowsPerDevice, n-1, n);
        }

        #pragma omp target update device(0) from(A2[mic0_halo_out : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) from(A2[mic1_halo_out : n]) nowait depend(inout:m1)

        #pragma omp taskwait
    }

    #pragma omp target update device(0) from(B2[mic0_out : dataLength])
    #pragma omp target update device(1) from(B2[mic1_out : dataLength])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
    int mic1_halo_out = rowsPerDevice * n;
    int m0, m1;

    start = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        #pragma omp target update device(0) to(A2[mic0_halo_in : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) to(A2[mic1_halo_in : n]) nowait depend(inout:m1)

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(0) nowait depend(inout:m0)
        {
            stencil_code_parallel(A2, B2, 1, rowsPerDevice, n);
        }

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(1) nowait depend(inout:m1)
        {
            stencil_code_parallel(A2, B2, rowsPerDevice, n-1, n);
        }

        #pragma omp target update device(0) from(A2[mic0_halo_out : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) from(A2[mic1_halo_out : n]) nowait depend(inout:m1)

        #pragma omp taskwait
    }

    #pragma omp target update device(0) from(B2[mic0_out : dataLength])
    #pragma omp target update device(1) from(B2[mic1_out : dataLength])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Alokujemy dane w pamięci koprocesorów

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
    int mic1_halo_out = rowsPerDevice * n;
    int m0, m1;

    start = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        #pragma omp target update device(0) to(A2[mic0_halo_in : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) to(A2[mic1_halo_in : n]) nowait depend(inout:m1)

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(0) nowait depend(inout:m0)
        {
            stencil_code_parallel(A2, B2, 1, rowsPerDevice, n);
        }

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(1) nowait depend(inout:m1)
        {
            stencil_code_parallel(A2, B2, rowsPerDevice, n-1, n);
        }

        #pragma omp target update device(0) from(A2[mic0_halo_out : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) from(A2[mic1_halo_out : n]) nowait depend(inout:m1)

        #pragma omp taskwait
    }

    #pragma omp target update device(0) from(B2[mic0_out : dataLength])
    #pragma omp target update device(1) from(B2[mic1_out : dataLength])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Alokujemy dane w pamięci koprocesorów

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
    int mic1_halo_out = rowsPerDevice * n;
    int m0, m1;

    start = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        #pragma omp target update device(0) to(A2[mic0_halo_in : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) to(A2[mic1_halo_in : n]) nowait depend(inout:m1)

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(0) nowait depend(inout:m0)
        {
            stencil_code_parallel(A2, B2, 1, rowsPerDevice, n);
        }

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(1) nowait depend(inout:m1)
        {
            stencil_code_parallel(A2, B2, rowsPerDevice, n-1, n);
        }

        #pragma omp target update device(0) from(A2[mic0_halo_out : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) from(A2[mic1_halo_out : n]) nowait depend(inout:m1)

        #pragma omp taskwait
    }

    #pragma omp target update device(0) from(B2[mic0_out : dataLength])
    #pragma omp target update device(1) from(B2[mic1_out : dataLength])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Wyznaczamy podział macierzy pomiędzy

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Alokujemy dane w pamięci koprocesorów

Wyznaczamy obszary komunikacji pomiędzy koprocesorami

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
    int mic1_halo_out = rowsPerDevice * n;
    int m0, m1;

    start = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        #pragma omp target update device(0) to(A2[mic0_halo_in : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) to(A2[mic1_halo_in : n]) nowait depend(inout:m1)

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(0) nowait depend(inout:m0)
        {
            stencil_code_parallel(A2, B2, 1, rowsPerDevice, n);
        }

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(1) nowait depend(inout:m1)
        {
            stencil_code_parallel(A2, B2, rowsPerDevice, n-1, n);
        }

        #pragma omp target update device(0) from(A2[mic0_halo_out : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) from(A2[mic1_halo_out : n]) nowait depend(inout:m1)

        #pragma omp taskwait
    }

    #pragma omp target update device(0) from(B2[mic0_out : dataLength])
    #pragma omp target update device(1) from(B2[mic1_out : dataLength])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Wyznaczamy podział macierzy pomiędzy

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Alokujemy dane w pamięci koprocesorów

Wyznaczamy obszary komunikacji pomiędzy koprocesorami

Uruchamiamy obliczenia dla zdefiniowanej liczby kroków czasowych oraz mierzymy czas

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
    int mic1_halo_out = rowsPerDevice * n;
    int m0, m1;

    start = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        #pragma omp target update device(0) to(A2[mic0_halo_in : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) to(A2[mic1_halo_in : n]) nowait depend(inout:m1)

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(0) nowait depend(inout:m0)
        {
            stencil_code_parallel(A2, B2, 1, rowsPerDevice, n);
        }

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(1) nowait depend(inout:m1)
        {
            stencil_code_parallel(A2, B2, rowsPerDevice, n-1, n);
        }

        #pragma omp target update device(0) from(A2[mic0_halo_out : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) from(A2[mic1_halo_out : n]) nowait depend(inout:m1)

        #pragma omp taskwait
    }

    #pragma omp target update device(0) from(B2[mic0_out : dataLength])
    #pragma omp target update device(1) from(B2[mic1_out : dataLength])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Wyznaczamy podział macierzy pomiędzy

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Alokujemy dane w pamięci koprocesorów

Wyznaczamy obszary komunikacji pomiędzy koprocesorami

Uruchamiamy obliczenia dla zdefiniowanej liczby kroków czasowych oraz mierzymy czas

Wymieniamy dane pomiędzy koprocesorami w każdym kroku czasowym

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
    int mic1_halo_out = rowsPerDevice * n;
    int m0, m1;

    start = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        #pragma omp target update device(0) to(A2[mic0_halo_in : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) to(A2[mic1_halo_in : n]) nowait depend(inout:m1)

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(0) nowait depend(inout:m0)
        {
            stencil_code_parallel(A2, B2, 1, rowsPerDevice, n);
        }

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(1) nowait depend(inout:m1)
        {
            stencil_code_parallel(A2, B2, rowsPerDevice, n-1, n);
        }

        #pragma omp target update device(0) from(A2[mic0_halo_out : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) from(A2[mic1_halo_out : n]) nowait depend(inout:m1)

        #pragma omp taskwait
    }

    #pragma omp target update device(0) from(B2[mic0_out : dataLength])
    #pragma omp target update device(1) from(B2[mic1_out : dataLength])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Wyznaczamy podział macierzy pomiędzy

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Alokujemy dane w pamięci koprocesorów

Wyznaczamy obszary komunikacji pomiędzy koprocesorami

Uruchamiamy obliczenia dla zdefiniowanej liczby kroków czasowych oraz mierzymy czas

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

Wymieniamy dane pomiędzy koprocesorami w każdym kroku czasowym

Po zakończeniu obliczeń przesyłamy dane do hosta

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
    int mic1_halo_out = rowsPerDevice * n;
    int m0, m1;

    start = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        #pragma omp target update device(0) to(A2[mic0_halo_in : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) to(A2[mic1_halo_in : n]) nowait depend(inout:m1)

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(0) nowait depend(inout:m0)
        {
            stencil_code_parallel(A2, B2, 1, rowsPerDevice, n);
        }

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(1) nowait depend(inout:m1)
        {
            stencil_code_parallel(A2, B2, rowsPerDevice, n-1, n);
        }

        #pragma omp target update device(0) from(A2[mic0_halo_out : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) from(A2[mic1_halo_out : n]) nowait depend(inout:m1)

        #pragma omp taskwait
    }

    #pragma omp target update device(0) from(B2[mic0_out : dataLength])
    #pragma omp target update device(1) from(B2[mic1_out : dataLength])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Wyznaczamy podział macierzy pomiędzy

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                A[i*n+j] = B[i*n+j];
            }
        }
    }
}
#pragma omp end declare target
```

Alokujemy dane w pamięci koprocesorów

Wyznaczamy obszary komunikacji pomiędzy koprocesorami

Uruchamiamy obliczenia dla zdefiniowanej liczby kroków czasowych oraz mierzymy czas

Obliczenia realizowane W ramach akceleratora są zrównoleglone z wykorzystaniem OpenMP

Wymieniamy dane pomiędzy koprocesorami w każdym kroku czasowym

Po zakończeniu obliczeń przesyłamy dane do hosta

Zwalniamy pamięć koprocesora

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
    int mic1_halo_out = rowsPerDevice * n;
    int m0, m1;

    start = omp_get_wtime();
    for(int ts=0; ts<timeSteps; ++ts)
    {
        #pragma omp target update device(0) to(A2[mic0_halo_in : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) to(A2[mic1_halo_in : n]) nowait depend(inout:m1)

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(0) nowait depend(inout:m0)
        {
            stencil_code_parallel(A2, B2, 1, rowsPerDevice, n);
        }

        #pragma omp target map(to: A2[0:n*n], B2[0:n*n]) device(1) nowait depend(inout:m1)
        {
            stencil_code_parallel(A2, B2, rowsPerDevice, n-1, n);
        }

        #pragma omp target update device(0) from(A2[mic0_halo_out : n]) nowait depend(inout:m0)
        #pragma omp target update device(1) from(A2[mic1_halo_out : n]) nowait depend(inout:m1)

        #pragma omp taskwait
    }

    #pragma omp target update device(0) from(B2[mic0_out : dataLength])
    #pragma omp target update device(1) from(B2[mic1_out : dataLength])
    stop = omp_get_wtime();
    cout << "Czas obliczen: " << stop-start << "\n\n";

    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Wyznaczamy podział macierzy pomiędzy

Algorytm: wersja 2 x Intel MIC

```
#pragma omp declare target
void stencil_code_parallel(double *A, double *B, const int iBegin, const int iEnd, const int n)
{
    #pragma omp parallel num_threads(240)
    {
        #pragma omp for schedule(static,1)
        for(int i=iBegin; i<iEnd; ++i)
        {
            for(int j=1; j<n-1; ++j)
            {
                B[i*n+j] = ( A[(i-1)*n+j] + A[i*n+(j-1)] + A[i*n+(j+1)] + A[(i+1)*n+j] ) * 0.25;
            }
        }

        #pragma omp for
        for(int i=iBegin,
        {
            for(int j=1;
            {
                A[i*n+j]
            }
        }
    }
}
#pragma omp end declare t
```

Alokujemy dane w pamięci koprocesorów

```
int main()
{
    //..
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target enter data map(to: A2[0:n*n], B2[0:n*n]) device(1)

    const int rowsPerDevice = n / 2;
    const int dataLength = rowsPerDevice * n;
    int mic0_out = 0;
    int mic1_out = dataLength;
    int mic0_halo_in = rowsPerDevice * n;
    int mic0_halo_out = (rowsPerDevice-1) * n;
    int mic1_halo_in = (rowsPerDevice-1) * n;
```

Wyznaczamy podział macierzy pomiędzy

Zasoby obliczeniowe	Czas obliczeń [s]	Przyspieszenie
2 x CPU	8,58	---
1 x Intel MIC	6,53	1,31x
2 x Intel MIC	3,86	2,22x

Kroki czasowe: 200, rozmiar macierzy: 10000x10000
Kompilator: icpc 17.0.1, flagi: -O3 -fopenmp
Platforma: 2 x Intel Xeon E5-2699 v3 + 2 x Intel Xeon Phi 7120P

Obliczenia re
W ramach ak
są zrównole
wykorzystanie

przesyłamy dane do hosta

Zwalniamy pamięć koprocesora

```
    //..
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(0)
    #pragma omp target exit data map(from: A2[0:n*n], B2[0:n*n]) device(1)
    //...
}
```

Zadanie 1

Przetestować we własnym zakresie przedstawione w prezentacji kody źródłowe.

Zadanie 2

Napisać program, który odpowiada za wyznaczenie sumę elementów dwóch macierzy w wymiarach *NxN.* Otrzymany wynik należy zapisać w trzeciej tablicy. Obliczenia realizowane w ramach aplikacji zrównoleglić wykorzystując jednocześnie zasoby dwóch akceleratorów Intel MIC. Wyniki obliczeń dla wersji hybrydowej porównać w wersją sekwencyjną realizowaną przez jeden wątek CPU.

Dziękuję za uwagę!