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 3

Środowisko OpenMP programowania maszyn z pamięcią wspólna, w tym procesorów wielordzeniowych

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

Politechnika Częstochowska

Standard OpenMP

- OpenMP (*Open Multi-Processing*) jest wieloplatformowym standardem programowania równoległego dedykowanym dla systemów z pamięcią współdzieloną
- OpenMP pozwala na tworzenie aplikacji równoległych dla systemów Linux oraz Windows
- Standard przeznaczony dla trzech języków programowania: C, C++ oraz Fortran

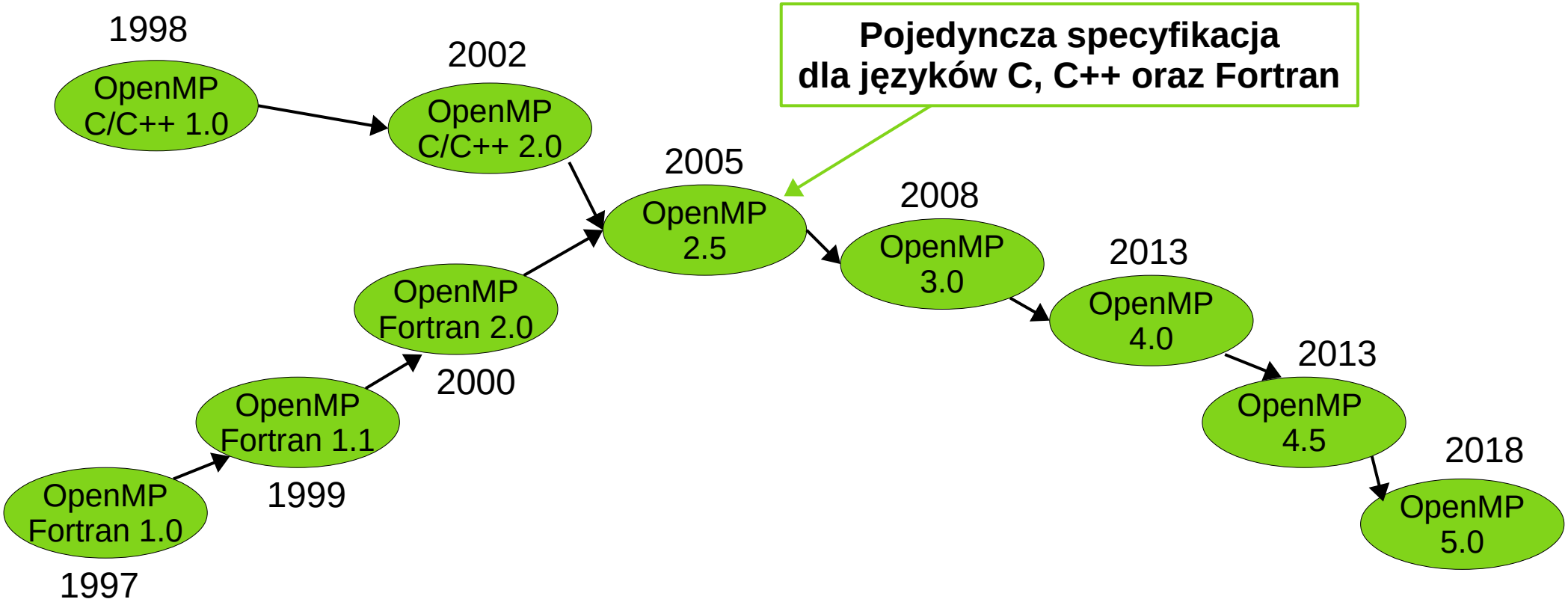
Źródło: www.openmp.org

Standard OpenMP

- Standard OpenMP składa się z trzech komponentów:
  - Dyrektyw kompilatora
  - Funkcji bibliotecznych
  - Zmiennych środowiskowych
- W skład komitetu pracującego nad standardem OpenMP wchodzą m.in. AMD, Convey, Cray, Fujitsu, HP, IBM, Intel, NEC, NVIDIA, Oracle, RedHat (GNU), ST Mircoelectronics
- Standard OpenMP jest stale rozwijany - aktualnie dostępną wersją standardu jest wersja 5.0
- Strona domowa: https://www.openmp.org

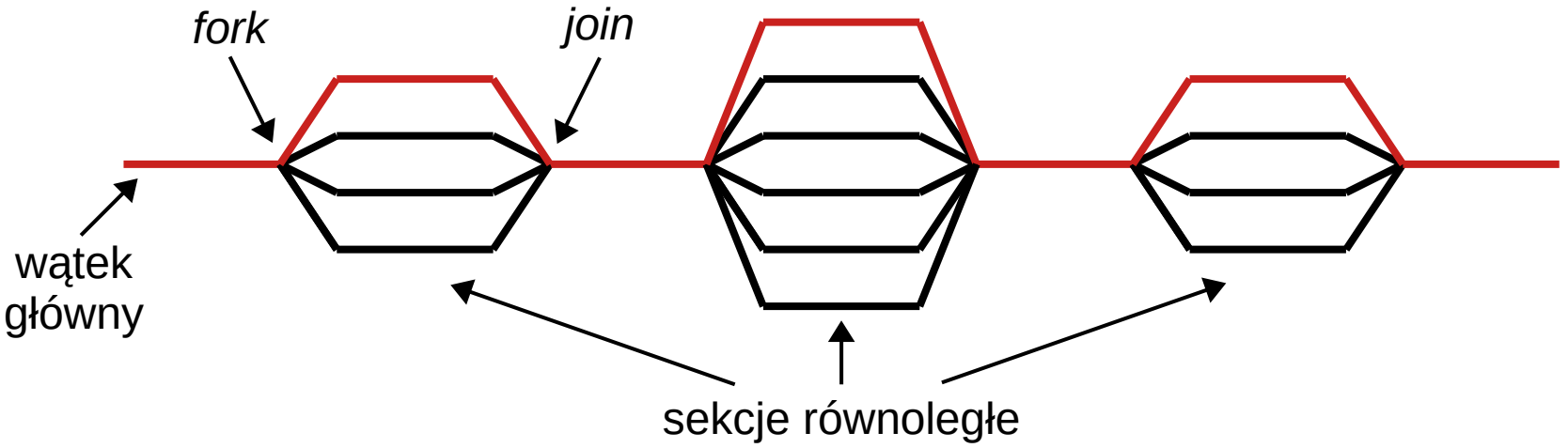
Źródło: www.openmp.org

Historia OpenMP



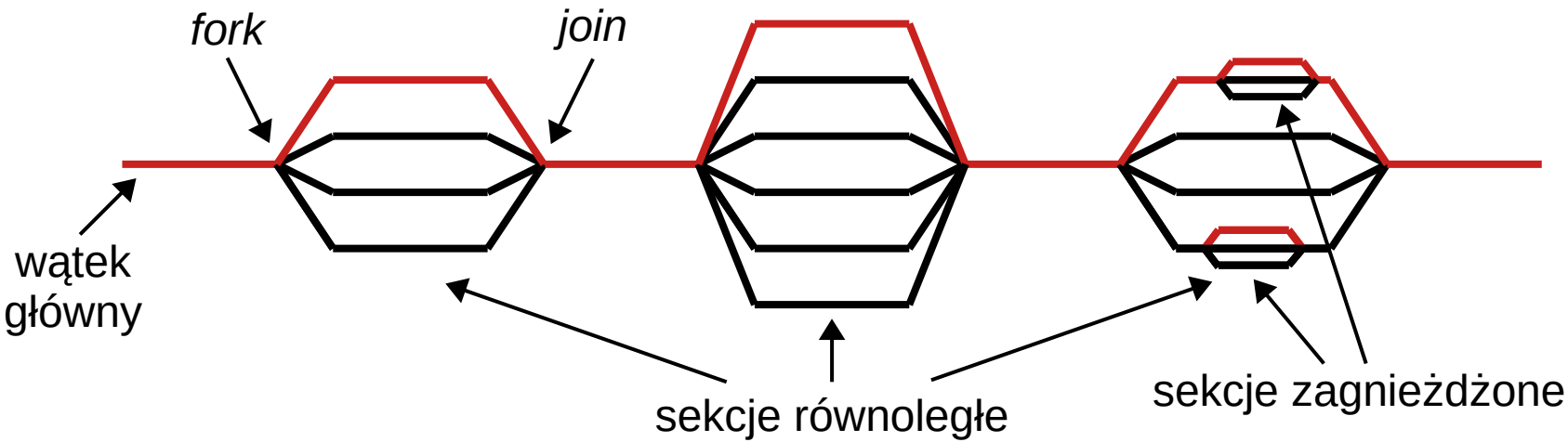
Model programowania

- Zrównoleglenie programu w standardzie OpenMP bazuje na modelu *fork-join*
  - Program wykonywany jest sekwencyjnie przez wątek główny
  - W momencie konieczności równoległej realizacji zadań tworzone są nowe wątki (*fork*)
  - Każdy wątek wykonuje określony zestaw instrukcji
  - Po zakończeniu zadań wątki są synchronizowane i usuwane (*join*), pozostaje jedynie wątek główny



Model programowania

- Standard OpenMP oferuje wsparcie dla zrównoleglenia zagnieżdżonego (ang. *nested parallelism*)
- Każdy z wątków w grupie może utworzyć własną grupę wątków
- Zagnieżdżone zrównoleglenie jest domyślnie wyłączone, aby z niego skorzystać należy ustawić odpowiednią zmienną systemową



Składania standardu OpenMP

- Do zrównoleglenia programu w standardzie OpenMP wykorzystuje się dyrektywy kompilatora
- Dla języków C/C++ dyrektywy OpenMP zawsze mają postać:

```
#pragma omp konstrukcja [klauzula1 [klazula2] …]
```

- Większość dyrektyw stosuje się do bloków kodu źródłowego (mogą być one również stosowane dla pojedynczych instrukcji)
- Funkcje biblioteczne zdefiniowane są w pliku nagłówkowym :

```
#include <omp.h>
```

Kompilacja programów OpenMP

- Kompilacja kodu źródłowego zawierającego dyrektywy OpenMP wymaga ustawienia dedykowanej flagi `-fopenmp`
  - W przypadku braku ustawionej flagi w trakcie kompilacji, dyrektywy powinny zostać zignorowane (zazwyczaj kompilator informuje o tym programistę za pomocą ostrzeżeń)
- Przykład kompilacji kodu źródłowego w systemach Linux z wykorzystaniem kompilatora GCC:

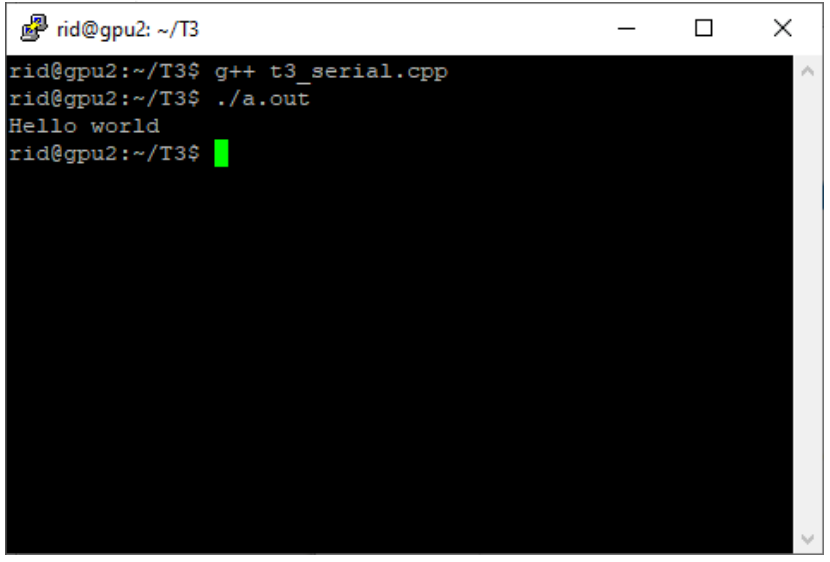
```
g++ -fopenmp program.cpp -o program
```

Zrównoleglenie aplikacji z wykorzystaniem standardu OpenMP

- Każdy program zrównoleglony przy pomocy standardu OpenMP posiada wątek główny działający w obrębie całego programu
- Program wykonywany jest sekwencyjnie do momentu napotkania dedykowanej konstrukcji OpenMP tworzącej nowe wątki:
  `#pragma omp parallel [klauzula1 [klazula2] …]`
- Dyrektywa ta informuje kompilator, że poprzedzony nią obszar kodu źródłowego zostanie wykonany w sposób równoległy
- Po zakończeniu wykonania regionu równoległego program wykonywany jest przez sekwencyjnie wątek główny
- Na końcu obszaru równoległego występuje niejawna synchronizacja wątków
- Liczba regionów równoległych w aplikacji może być dowolna, a każdy z nich może zostać wykonany przy użyciu innej liczby wątków

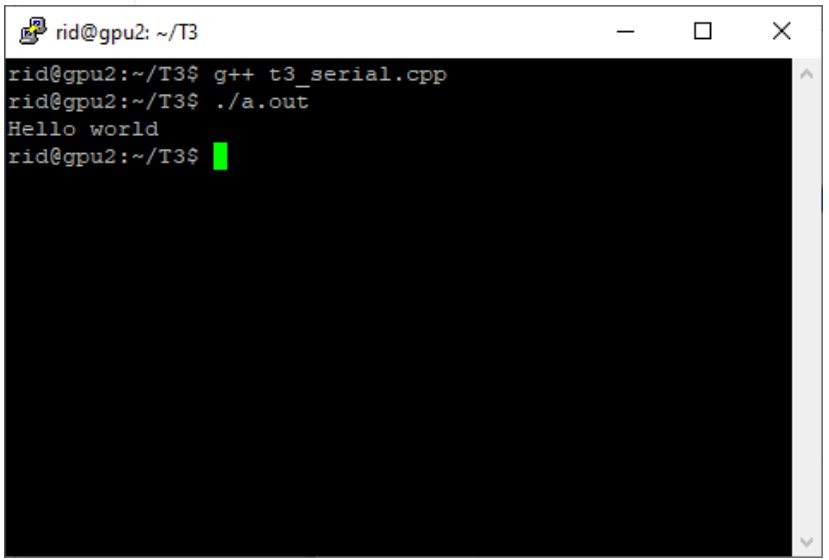
Przykład – Tworzenie wątków

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

int main()
{
    cout << "Hello world" << "\n";

    return 0;
}
```



Przykład – Tworzenie wątków

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

int main()
{
    cout << "Hello world" << "\n";

    return 0;
}
```

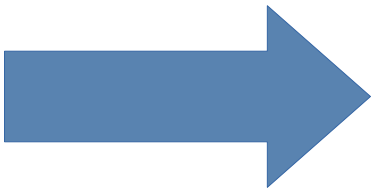


```cpp
#include <iostream>
using namespace std;

int main()
{
    #pragma omp parallel
    {
        cout << "Hello world" << "\n";
    }

    return 0;
}
```

Przykład – Tworzenie wątków

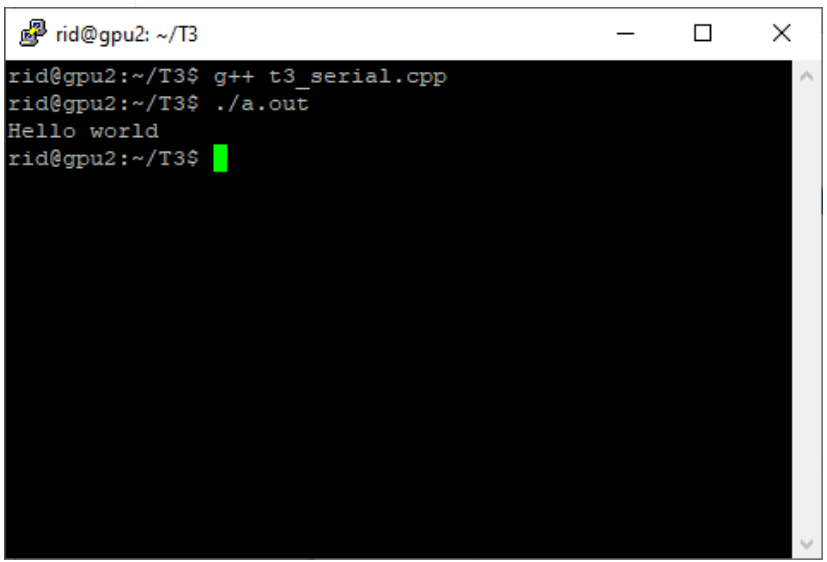
```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello world" << "\n";

    return 0;
}
```

Program zrównoleglony z wykorzystaniem OpenMP



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

int main()
{
    #pragma omp parallel
    {
        cout << "Hello world" << "\n";
    }

    return 0;
}
```

Region równoległy wykonywany przez wątki OpenMP



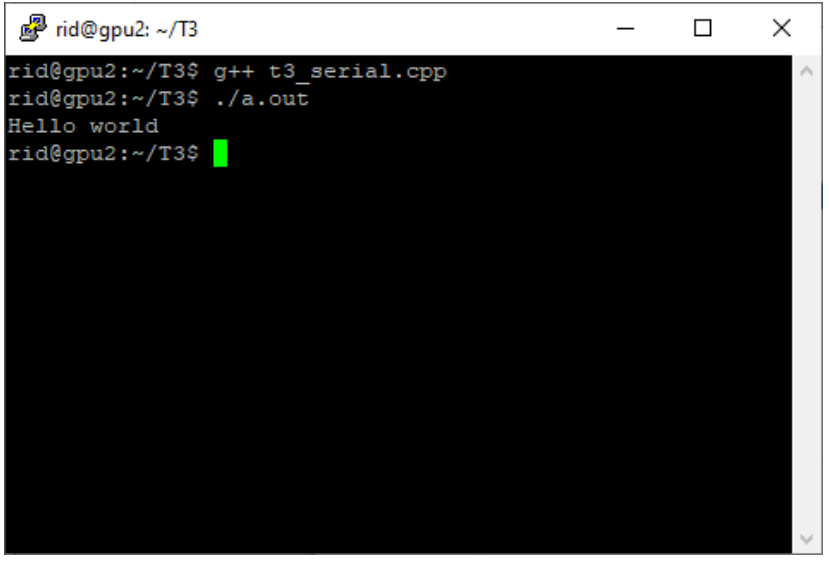
Przykład – Tworzenie wątków

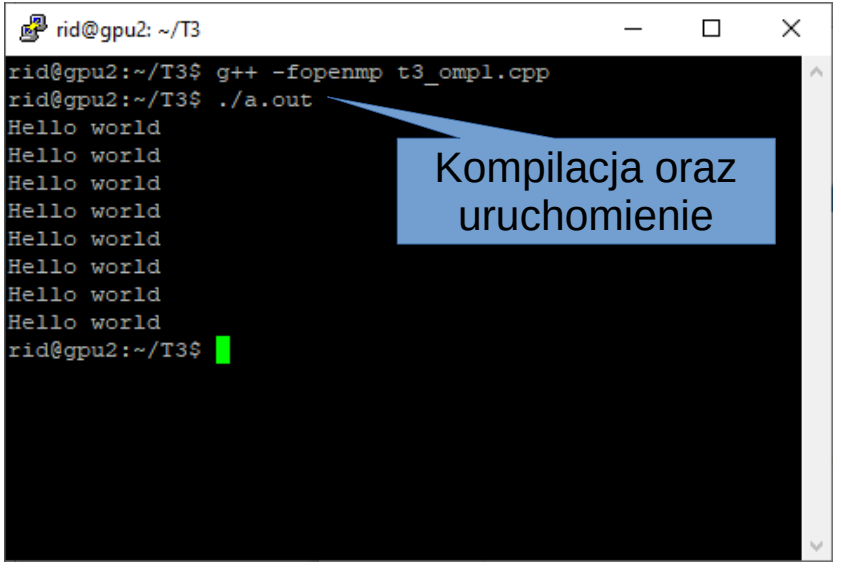
```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello world" << "\n";

    return 0;
}
```

Program zrównoleglony z wykorzystaniem OpenMP

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

int main()
{
    #pragma omp parallel
    {
        cout << "Hello world" << "\n";
    }
```

Region równoległy wykonywany przez wątki OpenMP





Kompilacja oraz uruchomienie

Definiowanie liczby wątków w grupie

- Zastosowanie dyrektywy `#pragma omp parallel` skutkuje utworzeniem wątków, których liczba jest równa liczbie logicznych rdzeni procesora
- Standard OpenMP dopuszcza możliwość określenia liczby tworzonych wątków przez programistę
- Osiągnięcie tego celu możliwe jest poprzez:
  - Użycie zmiennej środowiskowej `OMP_NUM_THREADS`
  - Wykorzystanie funkcji bibliotecznej `omp_set_num_threads()`
  - Zastosowanie klauzuli `num_threads()` dyrektywy `#pragma omp parallel`
- Istnieje również możliwość dynamicznego ustalania liczby wątków (aby np. umożliwić działanie dla systemów, które nie dysponują liczbą wątków określoną przykładowo za pomocą klauzuli `num_threads()`)

Przykład – Definiowanie liczby wątków za pomocą zmiennej środowiskowej

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

int main()
{
    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Przykład – Definiowanie liczby wątków za pomocą zmiennej środowiskowej

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

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

int main()
{
    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Przykład – Definiowanie liczby wątków za pomocą zmiennej środowiskowej

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

int main()
{
    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Tworzymy obszar równoległy

Przykład – Definiowanie liczby wątków za pomocą zmiennej środowiskowej

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

int main()
{
    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Tworzymy obszar równoległy

Funkcja `omp_get_thread_num()` zwaraca ID wątku w grupie

Przykład – Definiowanie liczby wątków za pomocą zmiennej środowiskowej

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

int main()
{
    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Tworzymy obszar równoległy

Funkcja omp_get_thread_num() zwaraca ID wątku w grupie

Funkcja omp_get_num_threads() zawraca aktualną ilość wątków w obszarze równoległym

Przykład – Definiowanie liczby wątków za pomocą zmiennej środowiskowej

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

int main()
{
    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Tworzymy obszar równoległy

Funkcja omp_get_thread_num() zwaraca ID wątku w grupie

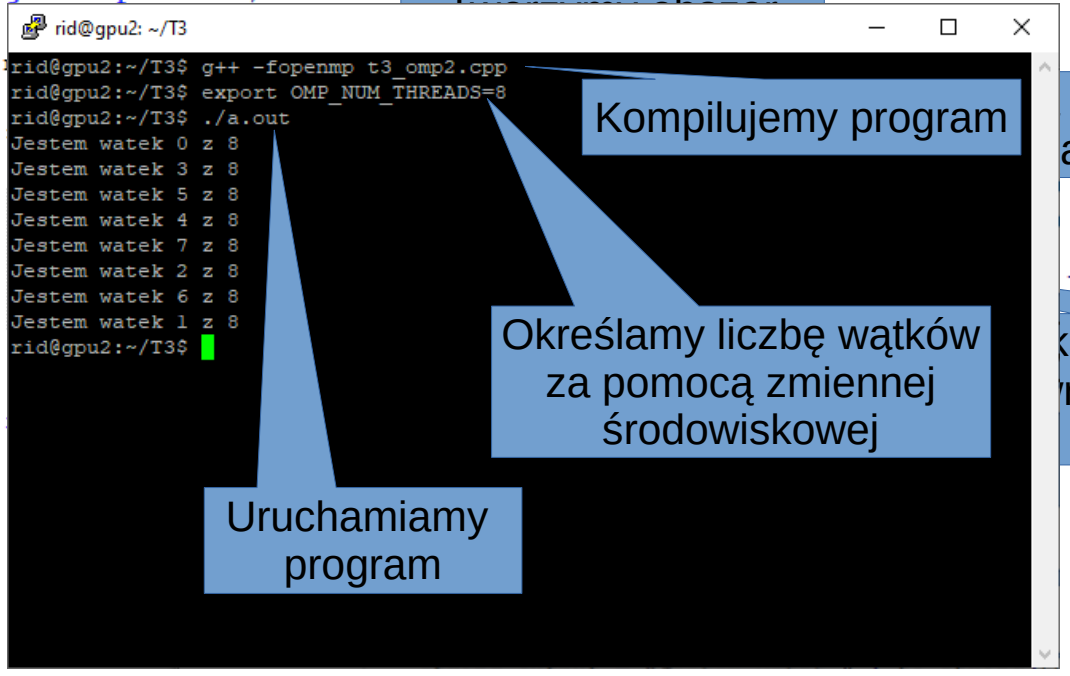
Funkcja omp_get_num_threads() zawraca aktualną ilość wątków w obszarze równoległym

Synchronizacja wątków

Przykład – Definiowanie liczby wątków za pomocą zmiennej środowiskowej

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

int
{

}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP



Kompilujemy program

Określamy liczbę wątków za pomocą zmiennej środowiskowej

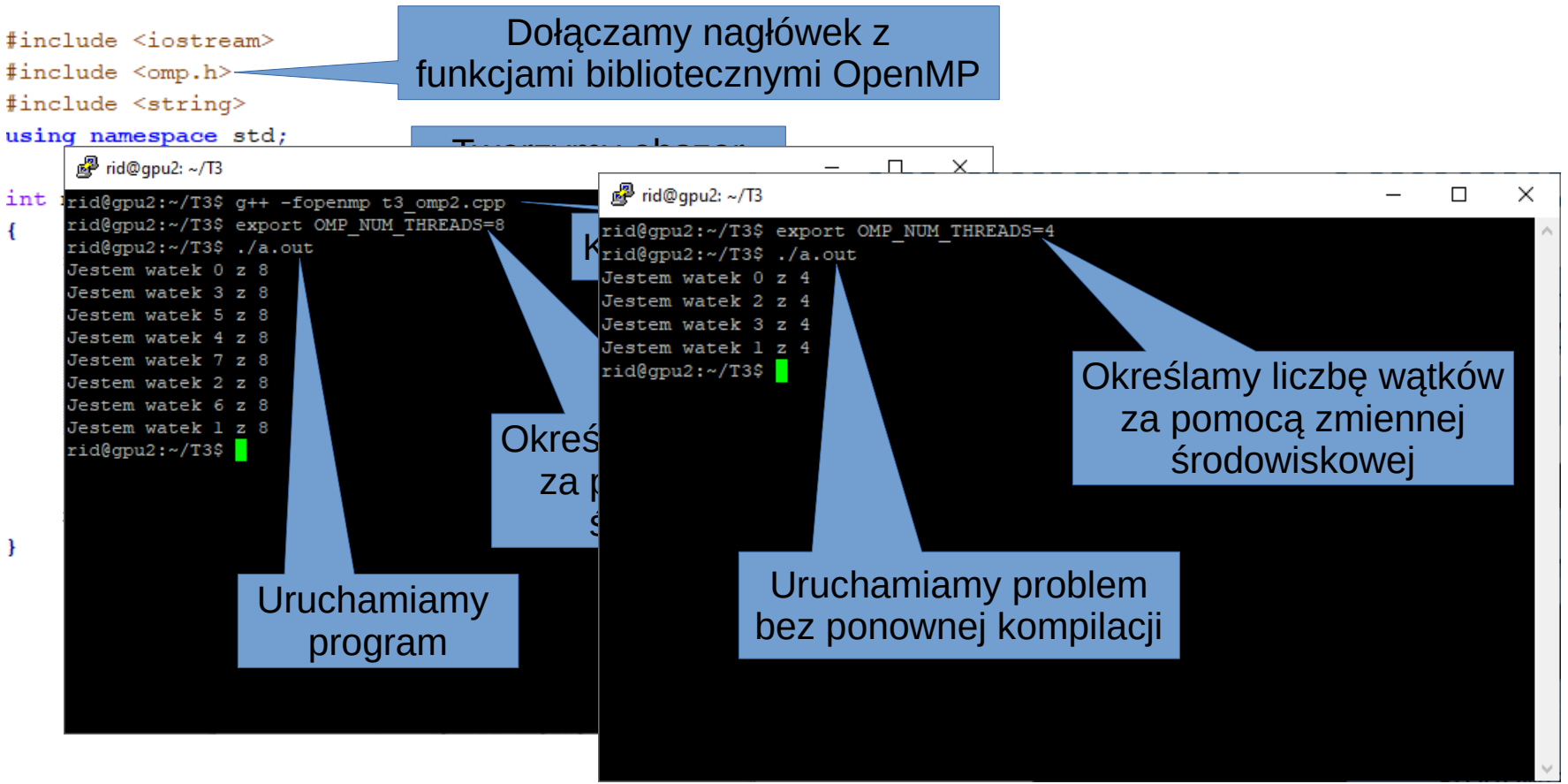
Uruchamiamy program

omp_get_thread_num()
aca ID wątku w grupie

+ to_string(totalThreads);

kcja omp_get_num_threads()
raca aktualną ilość wątków w obszarze równoległym

Przykład – Definiowanie liczby wątków za pomocą zmiennej środowiskowej



Przykład – Definiowanie liczby wątków za pomocą funkcji bibliotecznej

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

#define NUM_THREADS 5

int main()
{
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Przykład – Definiowanie liczby wątków za pomocą funkcji bibliotecznej

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Funkcja `omp_set_num_threads()` ustawia ilość wątków w grupie. Funkcja ta wywoływana jest na zewnątrz bloku `#pragma omp parallel`

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

#define NUM_THREADS 5

int main()
{
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Przykład – Definiowanie liczby wątków za pomocą funkcji bibliotecznej

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

#define NUM_THREADS 5

int main()
{
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Funkcja `omp_set_num_threads()` ustawia ilość wątków w grupie. Funkcja ta wywoływana jest na zewnątrz bloku `#pragma omp parallel`

Tworzymy obszar równoległy

Przykład – Definiowanie liczby wątków za pomocą funkcji bibliotecznej

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

#define NUM_THREADS 5

int main()
{
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Funkcja `omp_set_num_threads()` ustawia ilość wątków w grupie. Funkcja ta wywoływana jest na zewnątrz bloku `#pragma omp parallel`

Tworzymy obszar równoległy

Funkcja `omp_get_thread_num()` zwaraca ID wątku w grupie

Funkcja `omp_get_num_threads()` zawraca aktualną ilość wątków w obszarze równoległym

Przykład – Definiowanie liczby wątków za pomocą funkcji bibliotecznej

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

#define NUM_THREADS 5

int main()
{
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Funkcja `omp_set_num_threads()` ustawia ilość wątków w grupie. Funkcja ta wywoływana jest na zewnątrz bloku `#pragma omp parallel`

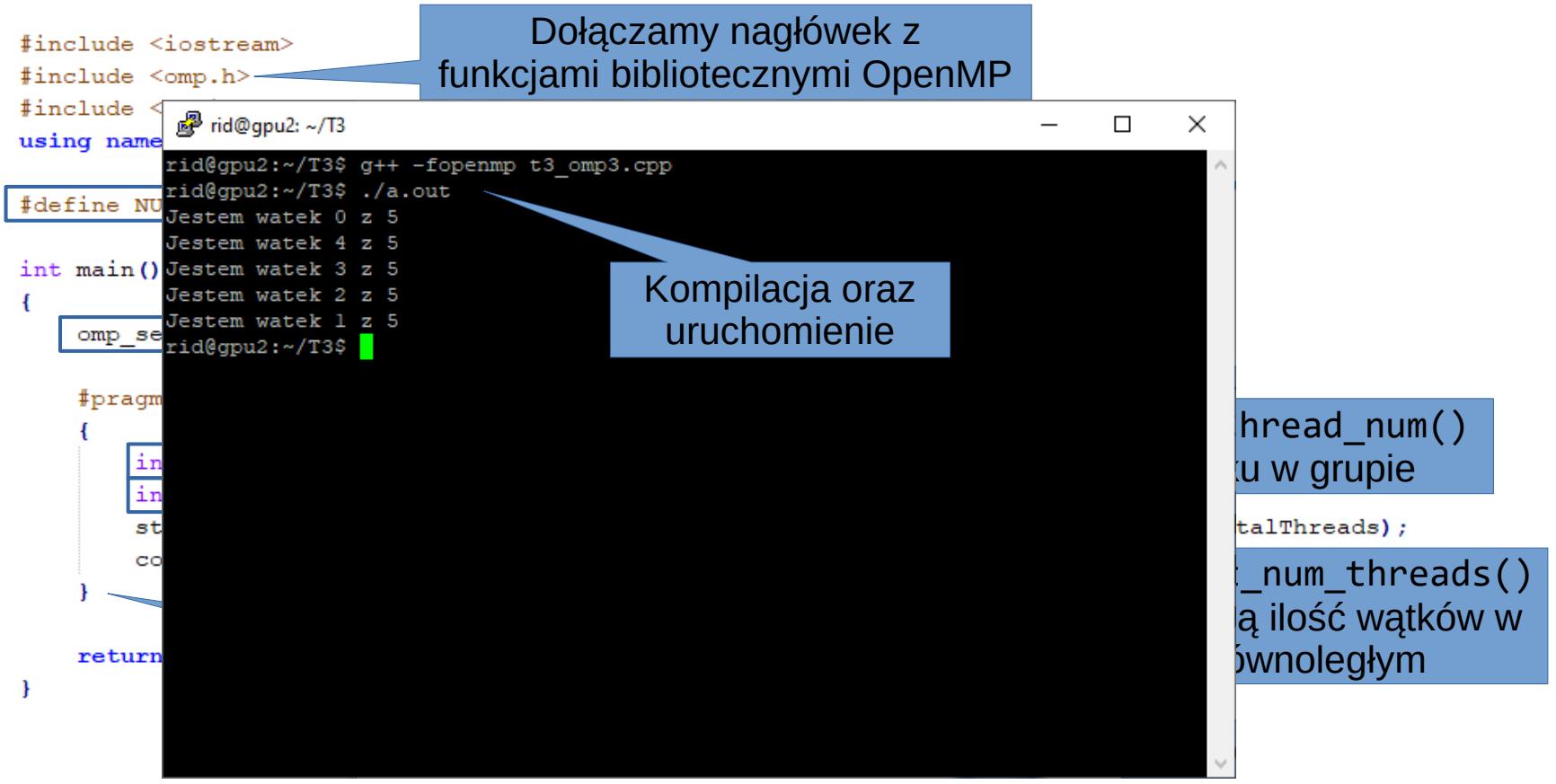
Tworzymy obszar równoległy

Funkcja `omp_get_thread_num()` zwaraca ID wątku w grupie

Funkcja `omp_get_num_threads()` zawraca aktualną ilość wątków w obszarze równoległym

Synchronizacja wątków

Przykład – Definiowanie liczby wątków za pomocą funkcji bibliotecznej



Przykład – Definiowanie liczby wątków za pomocą dedykowanej klauzuli

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

#define NUM_THREADS 5

int main()
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Przykład – Definiowanie liczby wątków za pomocą dedykowanej klauzuli

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

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

#define NUM_THREADS 5

int main()
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Przykład – Definiowanie liczby wątków za pomocą dedykowanej klauzuli

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Tworzymy obszar równoległy. Liczba wątków wewnątrz obszaru określona jest za pomocą klauzuli `num_threads`

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

#define NUM_THREADS 5

int main()
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " z " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Przykład – Definiowanie liczby wątków za pomocą dedykowanej klauzuli

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

#define NUM_THREADS 5

int main()
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Tworzymy obszar równoległy. Liczba wątków wewnątrz obszaru określona jest za pomocą klauzuli `num_threads`

Funkcja `omp_get_thread_num()` zwaraca ID wątku w grupie

Funkcja `omp_get_num_threads()` zawraca aktualną ilość wątków w obszarze równoległym

Przykład – Definiowanie liczby wątków za pomocą dedykowanej klauzuli

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

#define NUM_THREADS 5

int main()
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int threadID = omp_get_thread_num();
        int totalThreads = omp_get_num_threads();
        string output = "Jestem watek " + to_string(threadID) + " " + to_string(totalThreads);
        cout << output << "\n";
    }

    return 0;
}
```

Dołączamy nagłówek z funkcjami bibliotecznymi OpenMP

Tworzymy obszar równoległy. Liczba wątków wewnątrz obszaru określona jest za pomocą klauzuli `num_threads`

Funkcja `omp_get_thread_num()` zwaraca ID wątku w grupie

Funkcja `omp_get_num_threads()` zawraca aktualną ilość wątków w obszarze równoległym

Synchronizacja wątków

Przykład – Definiowanie liczby wątków za pomocą dedykowanej klauzuli

Dołączamy nagłówek z

```
#include <iostream>
#include <omp.h>
#include <string>
using namespace st

#define NUM_THREAD

int main()
{
    #pragma omp pa
    {
        int thread
        int totalT
        string out
        cout << ou
    }

    return 0;
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp4.cpp
rid@gpu2:~/T3$ ./a.out
Jestem watek 0 z 5
Jestem watek 4 z 5
Jestem watek 2 z 5
Jestem watek 1 z 5
Jestem watek 3 z 5
rid@gpu2:~/T3$
```

Kompilacja oraz uruchomienie

wnątrz
hreads

num()
bie

ds);

hreads()
wątków w
ym

Zrównoleglenie warunkowe

- Standard OpenMP dostarcza mechanizm pozwalający na utworzenie wątków, jedynie wtedy, kiedy spełniony jest określony warunek
- Wykorzystanie zrównoleglenia warunkowego wymaga zastosowania klauzuli `if()` dyrektywy `#pragma omp parallel`

  ```
  #pragma omp parallel if(wyrażenie całkowitoliczbowe)
  ```

- Blok kodu źródłowego wykonywany jest równolegle, wtedy i tylko wtedy gdy wyrażenie w nawiasach ma wartość różną od zera
  - W przeciwnym wypadku blok instrukcji wykonywany jest przez jeden wątek (wątek główny)

Przykład - Zrównoleglenie warunkowe

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

int main()
{
    int n;
    cout << "Rozmiar N: ";
    cin >> n;

    int *A = new int[n];
    int *B = new int[n];

    // Wypełnianie tablic
    /.../

    #pragma omp parallel num_threads(4) if(n > 1000)
    {
        // Obliczenia na tablicach A i B
        /.../
    }

    delete[] B;
    delete[] A;

    return 0;
}
```

Przykład - Zrównoleglenie warunkowe

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

int main()
{
    int n;
    cout << "Rozmiar N: ";
    cin >> n;

    int *A = new int[n];
    int *B = new int[n];

    // Wypełnianie tablic
    /.../

    #pragma omp parallel num_threads(4) if(n > 1000)
    {
        // Obliczenia na tablicach A i B
        /.../
    }

    delete[] B;
    delete[] A;

    return 0;
}
```

Wczytujemy z klawiatury wartość zmiennej n

Przykład - Zrównoleglenie warunkowe

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

int main()
{
    int n;
    cout << "Rozmiar N: ";
    cin >> n;

    int *A = new int[n];
    int *B = new int[n];

    // Wypełnianie tablic
    /.../

    #pragma omp parallel num_threads(4) if(n > 1000)
    {
        // Obliczenia na tablicach A i B
        /.../
    }

    delete[] B;
    delete[] A;

    return 0;
}
```

Wczytujemy z klawiatury wartość zmiennej n

Tworzymy oraz wypełniamy tablice, których rozmiar został pobrany od użytkownika

Przykład - Zrównoleglenie warunkowe

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

int main()
{
    int n;
    cout << "Rozmiar N: ";
    cin >> n;

    int *A = new int[n];
    int *B = new int[n];

    // Wypełnianie tablic
    /.../

    #pragma omp parallel num_threads(4) if(n > 1000)
    {
        // Obliczenia na tablicach A i B
        /.../
    }

    delete[] B;
    delete[] A;

    return 0;
}
```

Wczytujemy z klawiatury wartość zmiennej n

Tworzymy oraz wypełniamy tablice, których rozmiar został pobrany od użytkownika

Obliczenia na tablicach A i B wykonane zostaną równolegle przez **cztery wątki**, pod warunkiem, że ich rozmiar jest większy niż **1000 elementów**

Przykład - Zrównoleglenie warunkowe

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

int main()
{
    int n;
    cout << "Rozmiar N: ";
    cin >> n;

    int *A = new int[n];
    int *B = new int[n];

    // Wypełnianie tablic
    /.../

    #pragma omp parallel num_threads(4) if(n > 1000)
    {
        // Obliczenia na tablicach A i B
        /.../
    }

    delete[] B;
    delete[] A;

    return 0;
}
```

Wczytujemy z klawiatury wartość zmiennej n

Tworzymy oraz wypełniamy tablice, których rozmiar został pobrany od użytkownika

Obliczenia na tablicach A i B wykonane zostaną równolegle przez **cztery wątki**, pod warunkiem, że ich rozmiar jest większy niż **1000 elementów**

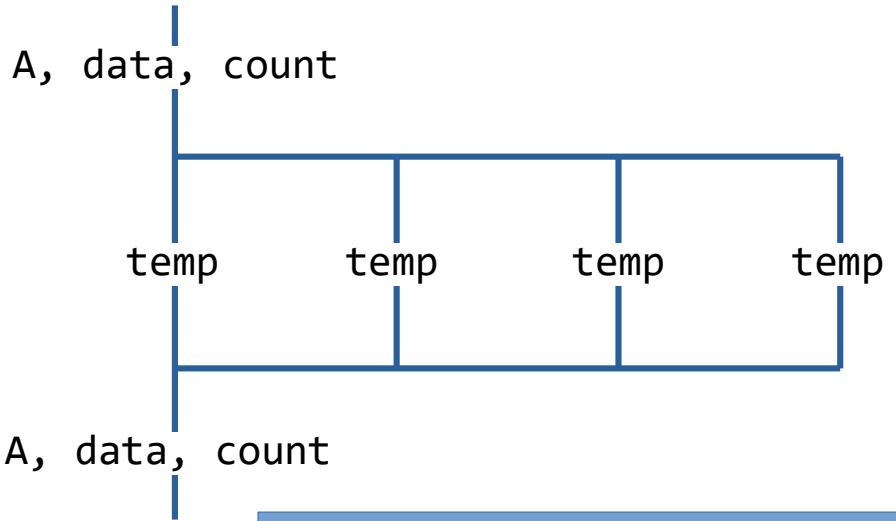
W przeciwnym wypadku, obliczenia te wykonane zostaną przez jeden wątek

Współdzielenie danych

- OpenMP jest interfejsem do tworzenia programów równoległych dla modelu ze współdzieloną pamięcią – komunikacja pomiędzy wątkami odbywa się poprzez **zmienne współdzielone**
- Każdy wątek w obszarze równoległym posiada zmienne prywatne i zmienne dzielone (wspólne dla wątków)
- Domyślnie każda zmienna widoczna podczas pracy równoległej jest traktowana jako wspólna dla wszystkich wątków
- Zmienne utworzone wewnątrz bloku równoległego są zmiennymi prywatnymi
- Zmienne statyczne zadeklarowane wewnątrz obszaru traktowane są jako wspólne
- Przykład zmiennych współdzielonych przez wątki:
  - Zmienne związane z obsługą plików
  - Dynamicznie przydzielona pamięć (`new`, `malloc`)
  - Stan generatora liczb losowych dla funkcji `rand()`

Współdzielenie danych

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

double A[10];

void work(const int *data)
{
    double temp[20];
    static int count;
   //.../
}

int main()
{
    int data[10];

    #pragma omp parallel
    {
        work(data);
        cout << A[0] << "\n";
    }

    return 0;
}
```



Zmienne A, data, count są współdzielone przez wszystkie wątki

Każdy wątek ma własną (prywatną) kopię tablicy temp

Widoczność zmiennych

- Standard OpenMP pozwala jawnie określić sposób dostępu przez wątki do zmiennych za pomocą dedykowanych klauzul dyrektywy `#pragma omp parallel`:
  - `shared(list)` – wyszczególnione zmienne są współdzielone przez wątki
  - `private(list)` – podane zmienne są prywatne dla każdego wątku
- Domyślnie wszystkie zmienne są współdzielone przez wątki (ang. *shared*)
- Można to nadpisać wykorzystując klauzulę `default`:
  - `default(shared)` – każda ze zmiennych widocznych w rozszerzeniu leksykalnym jest współdzielona
  - `default(none)` – żadna zmienna nie jest współdzielona
    - Dobra praktyka programistyczna, ponieważ musimy jawnie określić listę zmiennych współdzielonych oraz prywatnych
    - Pozwala to uniknąć potencjalnych sytuacji wyścigu do zmiennych

Klauzula `private`

- Zmienne zadeklarowane wewnątrz klauzuli `private` są prywatne dla każdego wątku (każdy wątek ma prywatną kopię zmiennych)
- Wartość zmiennych jest nieokreślona przy wejściu do obszaru równoległego, jak i po jego wyjściu

Klauzula `private`

- Zmienne zadeklarowane wewnątrz klauzuli `private` są prywatne dla każdego wątku (każdy wątek ma prywatną kopię zmiennych)
- Wartość zmiennych jest nieokreślona przy wejściu do obszaru równoległego, jak i po jego wyjściu

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) shared(a) private(b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Klauzula `private`

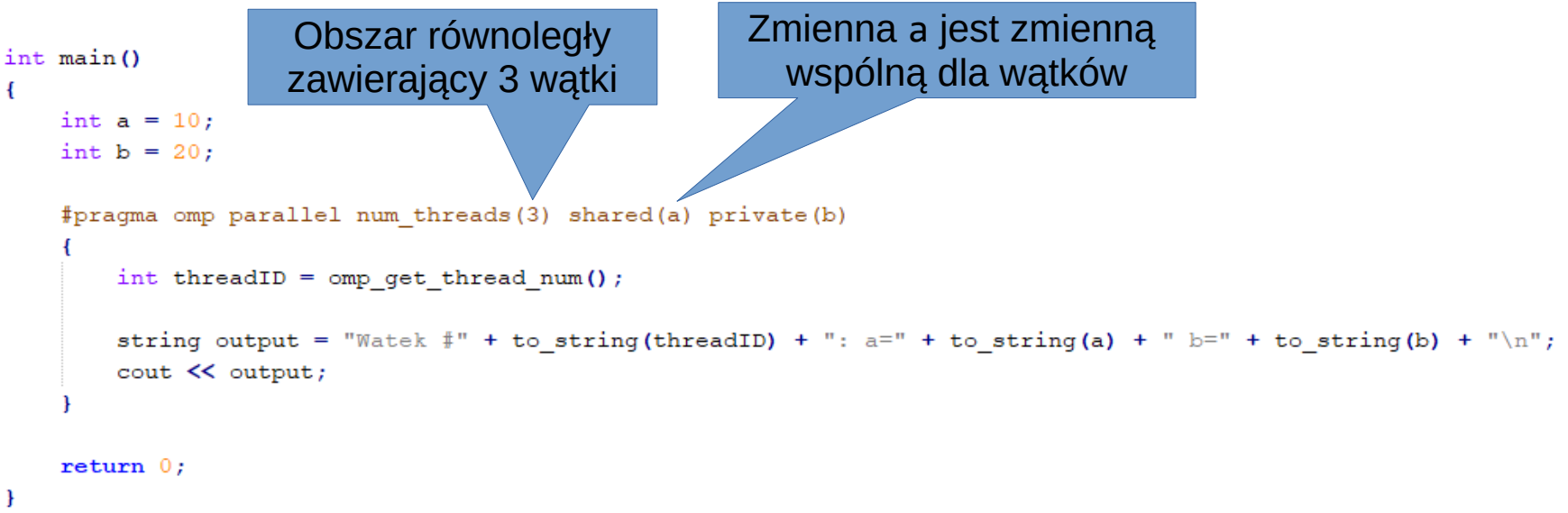
- Zmienne zadeklarowane wewnątrz klauzuli `private` są prywatne dla każdego wątku (każdy wątek ma prywatną kopię zmiennych)
- Wartość zmiennych jest nieokreślona przy wejściu do obszaru równoległego, jak i po jego wyjściu

Obszar równoległy zawierający 3 wątki

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) shared(a) private(b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

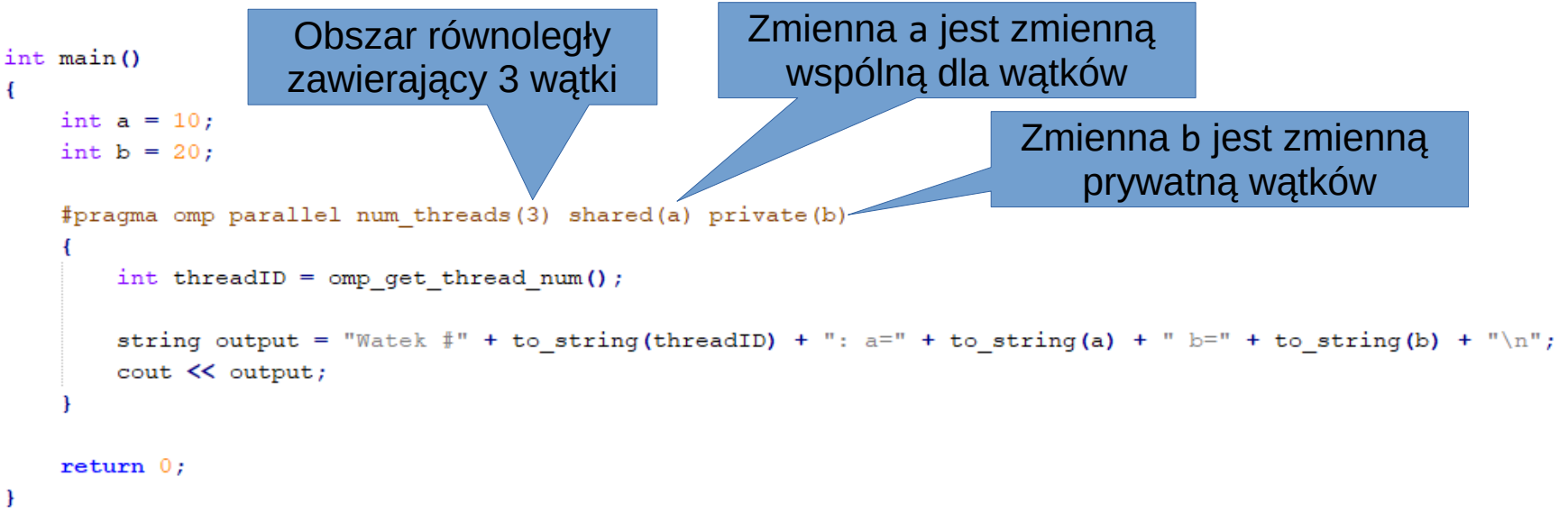
Klauzula `private`

- Zmienne zadeklarowane wewnątrz klauzuli `private` są prywatne dla każdego wątku (każdy wątek ma prywatną kopię zmiennych)
- Wartość zmiennych jest nieokreślona przy wejściu do obszaru równoległego, jak i po jego wyjściu



Klauzula `private`

- Zmienne zadeklarowane wewnątrz klauzuli `private` są prywatne dla każdego wątku (każdy wątek ma prywatną kopię zmiennych)
- Wartość zmiennych jest nieokreślona przy wejściu do obszaru równoległego, jak i po jego wyjściu



Klauzula `private`

- Zmienne zadeklarowane wewnątrz klauzuli `private` są prywatne dla każdego wątku (każdy wątek ma prywatną kopię zmiennych)
- Wartość zmiennych jest nieokreślona przy wejściu do obszaru równoległego, jak i po jego wyjściu

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) shared(a) private(b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Obszar równoległy zawierający 3 wątki

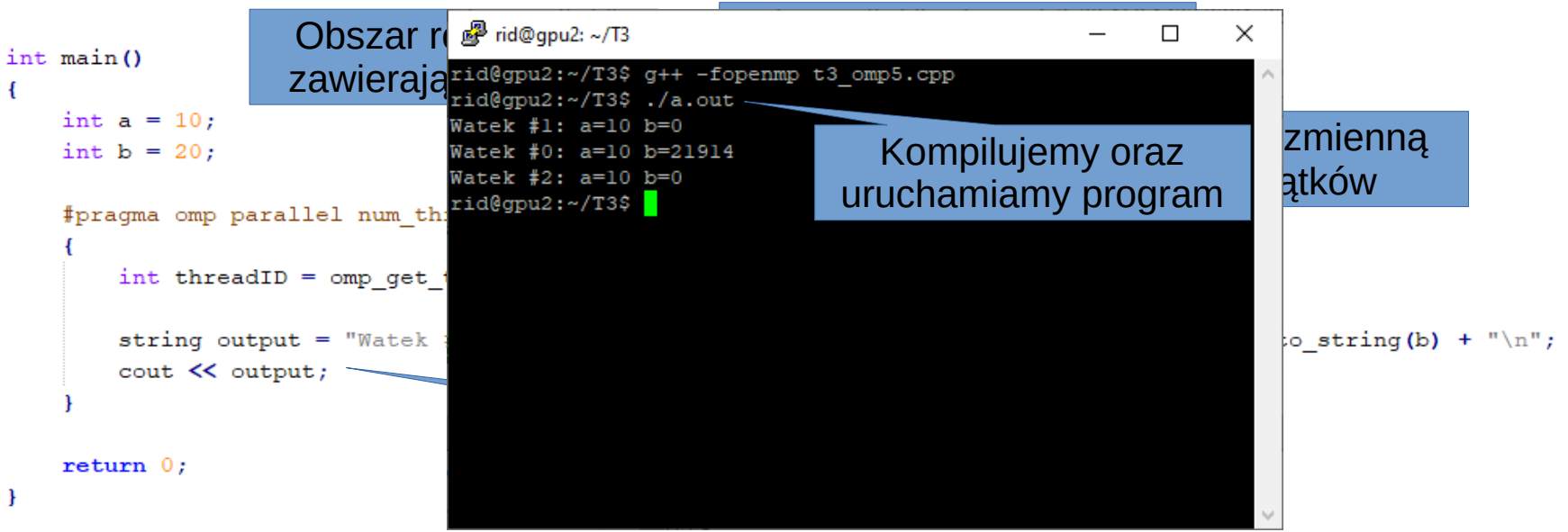
Zmienna a jest zmienną wspólną dla wątków

Zmienna b jest zmienną prywatną wątków

Wyświetlamy wartości zmiennych a oraz b w ramach wątków

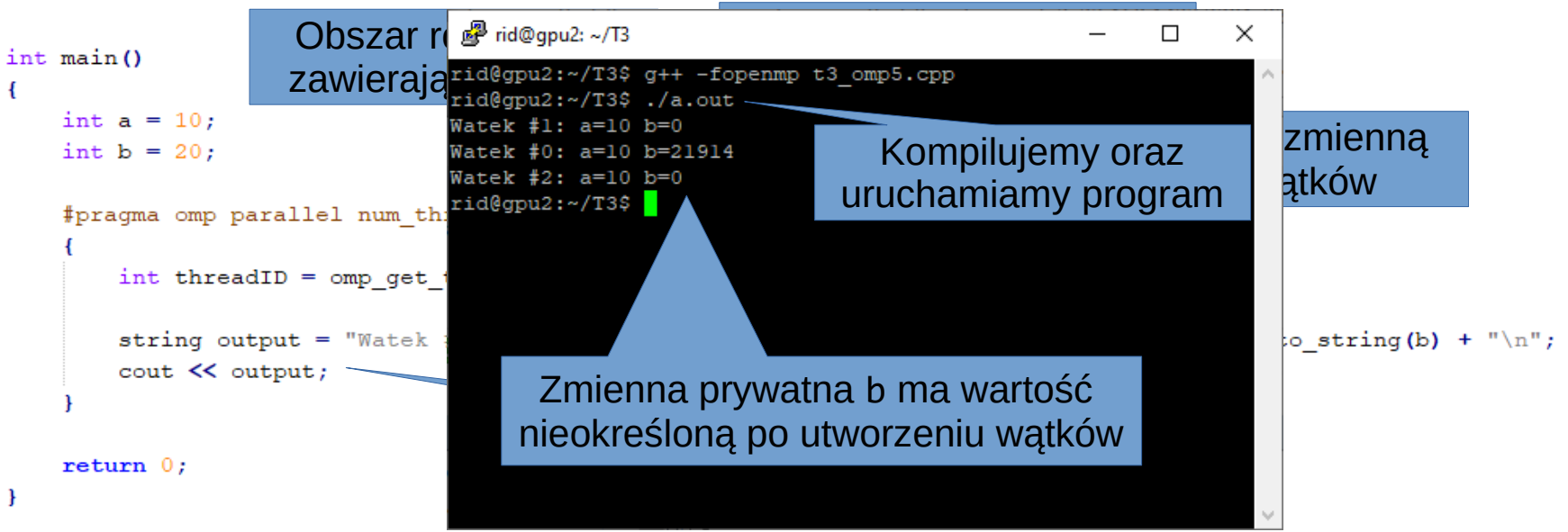
Klauzula `private`

- Zmienne zadeklarowane wewnątrz klauzuli `private` są prywatne dla każdego wątku (każdy wątek ma prywatną kopię zmiennych)
- Wartość zmiennych jest nieokreślona przy wejściu do obszaru równoległego, jak i po jego wyjściu



Klauzula `private`

- Zmienne zadeklarowane wewnątrz klauzuli `private` są prywatne dla każdego wątku (każdy wątek ma prywatną kopię zmiennych)
- Wartość zmiennych jest nieokreślona przy wejściu do obszaru równoległego, jak i po jego wyjściu



Klauzula `firstprivate`

- Klauzula `firstprivate` jest rozszerzeniem klauzuli `private`
- Tworzone zmienne prywatne inicjalizowane są wartością zmiennej globalnej

Klauzula `firstprivate`

- Klauzula `firstprivate` jest rozszerzeniem klauzuli `private`
- Tworzone zmienne prywatne inicjalizowane są wartością zmiennej globalnej

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) private(a) firstprivate(b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Klauzula `firstprivate`

- Klauzula `firstprivate` jest rozszerzeniem klauzuli `private`
- Tworzone zmienne prywatne inicjalizowane są wartością zmiennej globalnej

Obszar równoległy zawierający 3 wątki

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) private(a) firstprivate(b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Klauzula `firstprivate`

- Klauzula `firstprivate` jest rozszerzeniem klauzuli `private`
- Tworzone zmienne prywatne inicjalizowane są wartością zmiennej globalnej

Obszar równoległy zawierający 3 wątki

Zmienne a oraz b są prywatne dla wątków

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) private(a) firstprivate(b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Klauzula `firstprivate`

- Klauzula `firstprivate` jest rozszerzeniem klauzuli `private`
- Tworzone zmienne prywatne inicjalizowane są wartością zmiennej globalnej

Obszar równoległy zawierający 3 wątki

Zmienne a oraz b są prywatne dla wątków

Zmienna a jest zadeklarowana z wykorzystaniem klauzuli `private`

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) private(a) firstprivate(b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Klauzula `firstprivate`

- Klauzula `firstprivate` jest rozszerzeniem klauzuli `private`
- Tworzone zmienne prywatne inicjalizowane są wartością zmiennej globalnej

Obszar równoległy zawierający 3 wątki

Zmienne a oraz b są prywatne dla wątków

Zmienna a jest zadeklarowana z wykorzystaniem klauzuli `private`

Zmienna b zadeklarowana jest przy użyciu klauzuli `firstprivate`

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) private(a) firstprivate(b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Klauzula `firstprivate`

- Klauzula `firstprivate` jest rozszerzeniem klauzuli `private`
- Tworzone zmienne prywatne inicjalizowane są wartością zmiennej globalnej

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) p
    {
        int threadID = omp_get_thread_num

        string output = "Watek #" + to_st
        cout << output;
    }

    return 0;
}
```

Obszar równoległy zawierający 3 wątki

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp6.cpp
rid@gpu2:~/T3$ ./a.out
Watek #0: a=21881 b=20
Watek #1: a=0 b=20
Watek #2: a=0 b=20
rid@gpu2:~/T3$
```

Kompilujemy oraz uruchamiamy program

Klauzula `firstprivate`

- Klauzula `firstprivate` jest rozszerzeniem klauzuli `private`
- Tworzone zmienne prywatne inicjalizowane są wartością zmiennej globalnej

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) p
    {
        int threadID = omp_get_thread_num

        string output = "Watek #" + to_st
        cout << output;
    }

    return 0;
}
```

Obszar równoległy zawierający 3 wątki

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp6.cpp
rid@gpu2:~/T3$ ./a.out
Watek #0: a=21881 b=20
Watek #1: a=0 b=20
Watek #2: a=0 b=20
rid@gpu2:~/T3$
```

Kompilujemy oraz uruchamiamy program

Zmienna prywatna b `wątków` ma wartość zgodną z pierwowzorem w wątku głównym

Klauzula `firstprivate`

- Klauzula `firstprivate` jest rozszerzeniem klauzuli `private`
- Tworzone zmienne prywatne inicjalizowane są wartością zmiennej globalnej

```
int main()
{
    int a = 10;
    int b = 20;

    #pragma omp parallel num_threads(3) p
    {
        int threadID = omp_get_thread_num

        string output = "Watek #" + to_st
        cout << output;
    }

    return 0;
}
```

Obszar równoległy zawierający 3 wątki

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp6.cpp
rid@gpu2:~/T3$ ./a.out
Watek #0: a=21881 b=20
Watek #1: a=0 b=20
Watek #2: a=0 b=20
rid@gpu2:~/T3$
```

Kompilujemy oraz uruchamiamy program

Zmienna prywatna b `wątków` ma wartość zgodną z pierwowzorem w wątku głównym

Zmienna prywatna a ma wartość nieokreśloną po utworzeniu wątków

Dyrektywa `threadprivate`

- Dyrektywa `threadprivate` pozwala na definiowanie zmiennych globalnych, dla których każdy wątek będzie miał własną kopię

  ```
  #pragma omp threadprivate(list)
  ```

- Wszystkie zmienne podane jako parametry będą prywatne dla wątków w każdym bloku równoległym

Dyrektywa threadprivate

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";


    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";


    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";


    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

Pierwszy obszar równoległy

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";


    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

Domyślnie wszystkie zmienne nie są współdzielone

Pierwszy obszar równoległy

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";


    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

Domyślnie wszystkie zmienne nie są współdzielone

Pierwszy obszar równoległy

Jedynie tablica data jest współdzielona pomiędzy wątki

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";


    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

Domyślnie wszystkie zmienne nie są współdzielone

Pierwszy obszar równoległy

Jedynie tablica data jest współdzielona pomiędzy wątki

Każdy z wątków zapisuje wartość zmiennej a do tablicy data pod indeksem równym ID wątku

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";


    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

Domyślnie wszystkie zmienne nie są współdzielone

Pierwszy obszar równoległy

Jedynie tablica data jest współdzielona pomiędzy wątki

Każdy z wątków zapisuje wartość zmiennej a do tablicy data pod indeksem równym ID wątku

Modyfikujemy wartość zmiennej a

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

Domyślnie wszystkie zmienne nie są współdzielone

Pierwszy obszar równoległy

Jedynie tablica data jest współdzielona pomiędzy wątki

Wyświetlamy zawartość tablicy data

Każdy z wątków zapisuje wartość zmiennej a do tablicy data pod indeksem równym ID wątku

Modyfikujemy wartość zmiennej a

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

Domyślnie wszystkie zmienne nie są współdzielone

Pierwszy obszar równoległy

Jedynie tablica data jest współdzielona pomiędzy wątki

Wyświetlamy zawartość tablicy data

Każdy z wątków zapisuje wartość zmiennej a do tablicy data pod indeksem równym ID wątku

Modyfikujemy wartość zmiennej a

Drugi obszar równoległy

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

Domyślnie wszystkie zmienne nie są współdzielone

Pierwszy obszar równoległy

Jedynie tablica data jest współdzielona pomiędzy wątki

Wyświetlamy zawartość tablicy data

Każdy z wątków zapisuje wartość zmiennej a do tablicy data pod indeksem równym ID wątku

Modyfikujemy wartość zmiennej a

Drugi obszar równoległy

Ponownie zapisujemy wartość zmiennej a do tablicy data

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która będzie zmienną prywatną wątków w obrębie całego programu

Domyślnie wszystkie zmienne nie są współdzielone

Pierwszy obszar równoległy

Jedynie tablica data jest współdzielona pomiędzy wątki

Wyświetlamy zawartość tablicy data

Każdy z wątków zapisuje wartość zmiennej a do tablicy data pod indeksem równym ID wątku

Modyfikujemy wartość zmiennej a

Drugi obszar równoległy

Raz jeszcze wyświetlamy zawartość tablicy data

Ponownie zapisujemy wartość zmiennej a do tablicy data

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

#define NUM_THREADS 5

int a = 20;
#pragma omp threadprivate(a)

int main()
{
    int data[NUM_THREADS];

    cout << "Parallel region NO.: 1" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
        a = threadID * 10;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    cout << "\n" << "Parallel region NO.: 2" << "\n";
    #pragma omp parallel num_threads(NUM_THREADS) default(none) shared(data)
    {
        int threadID = omp_get_thread_num();
        data[threadID] = a;
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << data[i] << "\n";

    return 0;
}
```

Dyrektywa threadprivate

Liczba wątków

Tworzymy zmienną a, która jest zmienną prywatną wątków w obrębie całego programu

```
#include <iostream>
#include <omp.h>
#include <string>
```

rid@gpu2: ~/T3

```
rid@gpu2:~/T3$ g++ -fopenmp t3_omp7.cpp
rid@gpu2:~/T3$ ./a.out
Parallel region NO.: 1
20
20
20
20
20

Parallel region NO.: 2
0
10
20
30
40
rid@gpu2:~/T3$
```

Kompilujemy oraz uruchamiamy program

Początkowo wartość zmiennej prywatnej odpowiada jej pierwowzorowi w wątku głównym

Wartość zmiennej prywatnej nie ulega zmianie pomiędzy obszarami równoległymi

Pierwszy obszar równoległy

Wyświetlamy zawartość tablicy data

Raz jeszcze wyświetlamy zawartość tablicy data

ą współdzielone

tablica data jest lona pomiędzy wątki

e wartość zmiennej a do ksem równym ID wątku

obszar równoległy

ść

a

```
    return 0;
}
```

Serializacja w obszarze równoległym

- Standard OpenMP dostarcza konstrukcje pozwalające na serializacje zadań w obszarze równoległym
- Wykonanie pewnych fragmentów kodu programu w sekcji równoległej tylko przez jeden wątek możliwe jest poprzez zastosowanie następujących dyrektyw:
  - `#pragma omp master`
  - `#pragma omp single`

Dyrektywa `omp master`

- Dyrektywa `omp master` wymusza wykonanie bloku kodu tylko przez **wątek główny** (wątek o identyfikatorze równym 0)
- Pozostałe wątki pomijają blok oznaczony tą dyrektywą i wykonują pozostały kod w obszarze równoległym
- Wątki nie są blokowane po sekcji wykonywanej przez wątek główny
- Nie wymaga pobierania, ani sprawdzania identyfikatora wątku
- Przydatna przykładowo w przypadku konieczności wykonania operacji wejścia-wyjścia w obszarze równoległym

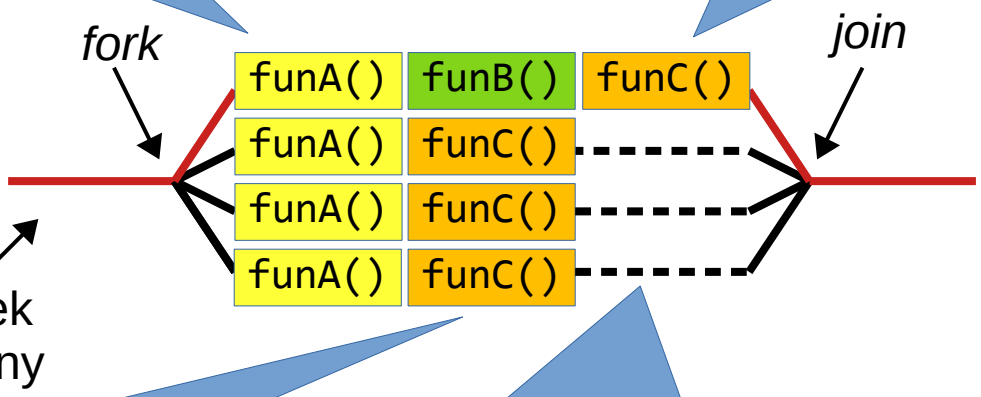
Dyrektywa omp master

- Przykład:

```
int main()
{
    #pragma omp parallel num_threads(4)
    {
        funA();

        #pragma omp master
        {
            funB();
        }

        funC();
    }

    return 0;
}
```

Funkcja funA wykonywana jest jednocześnie przez wszystkie wątki

Po wykonaniu kodu funkcji funB wątek główny rozpoczyna realizację funkcji funC



Funkcja funB realizowana jest jedynie przez wątek główny. W tym samym czasie pozostałe wątki wykonują funkcje funC.

Jednocześnie, pozostałe wątki oczekują na wątek główny

Dyrektywa `omp single`

- Stosowana do wymuszenia wykonania wskazanego bloku tylko przez **jeden** wątek (niekoniecznie główny)
- Dyrektywa `single` nie określa, który z wątków wykona blok kodu źródłowego
- Pozostałe wątki pomijają instrukcje w bloku i oczekują na jego końcu

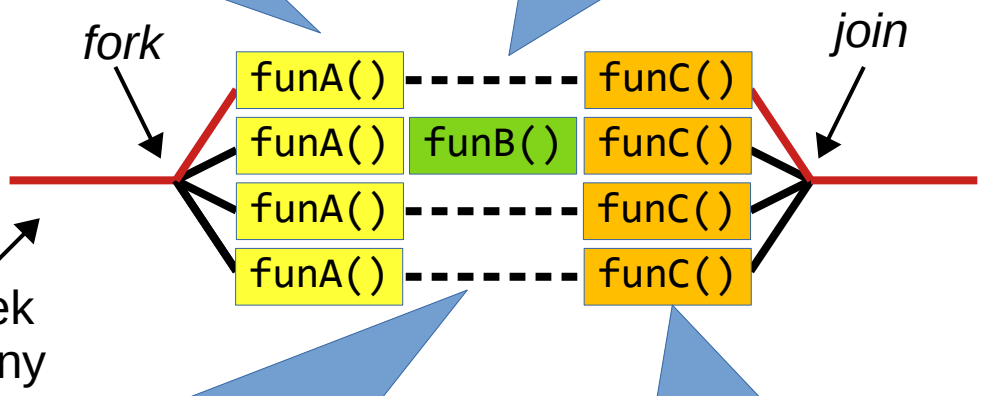
Dyrektywa omp single

- Przykład:

```
int main()
{
    #pragma omp parallel num_threads(4)
    {
        funA();

        #pragma omp single
        {
            funB();
        }

        funC();
    }

    return 0;
}
```

Funkcja funA wykonywana jest jednocześnie przez wszystkie wątki

Funkcja funB realizowana jest przez jeden z wątków w grupie



Pozostałe wątki oczekują, aż dany wątek zakończy realizację funkcji funB

Po zakończeniu realizacji bloku funB wszystkie wątki rozpoczynają obliczenia w ramach funkcji funC

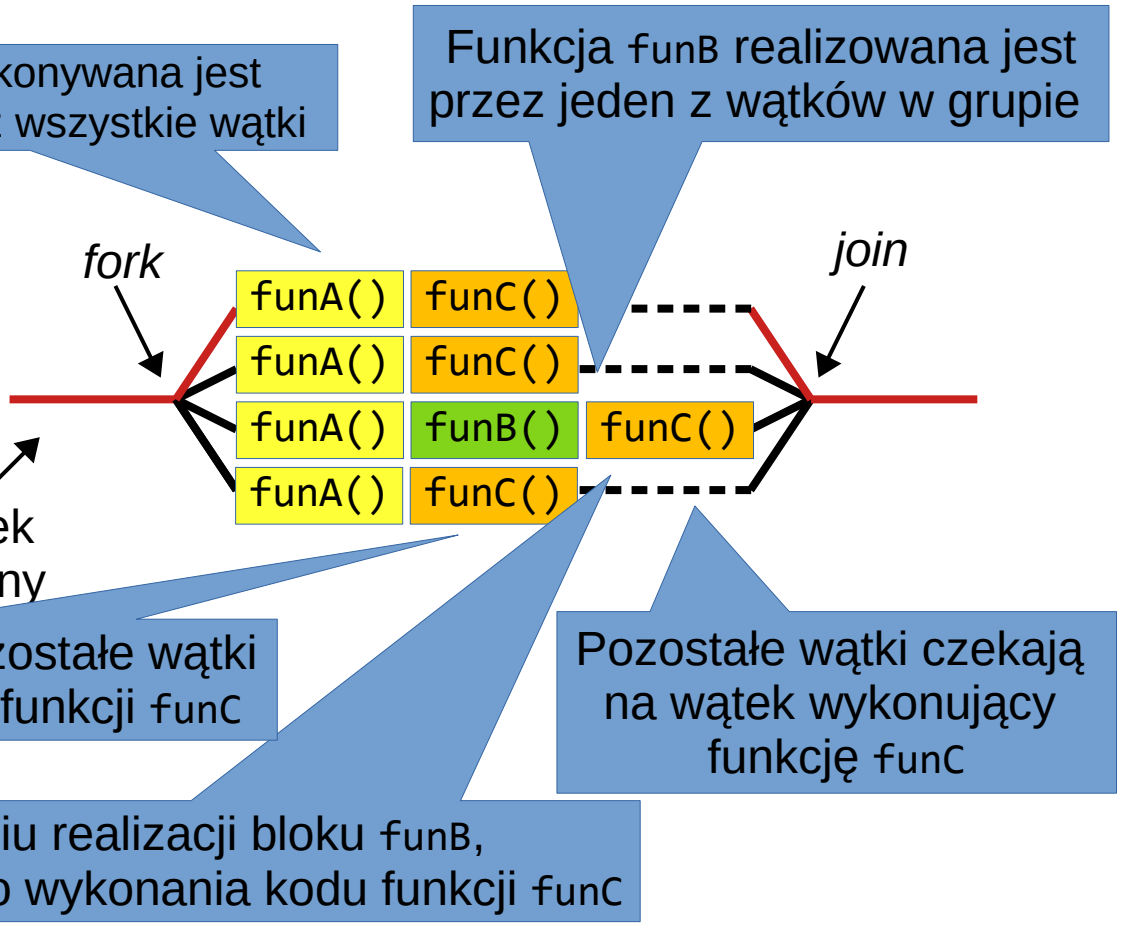
Dyrektywa `omp single`

- Stosowana do wymuszenia wykonania wskazanego bloku tylko przez **jeden** wątek (niekoniecznie główny)
- Dyrektywa `single` nie określa, który z wątków wykona blok kodu źródłowego
- Pozostałe wątki pomijają instrukcje w bloku i oczekują na jego końcu
- Standard OpenMP dostarcza klauzulę `nowait` pozwalającą usunąć konieczność blokowania wątków do czasu zakończenia realizacji instrukcji w bloku `single`

Dyrektywa omp single

- Przykład:

```
int main()
{
    #pragma omp parallel num_threads(4)
    {
        funA();

        #pragma omp single nowait
        {
            funB();
        }

        funC();
    }

    return 0;
}
```



Dyrektywa `omp single`

- Dyrektywa omp `single` posiada także przydatną klauzulę `copyprivate`
- Wykorzystywana jest do rozgłaszania (ang. *broadcast*) wartości zmiennych lokalnych wątku, który wykonuje sekcję `single` do pozostałych wątków
- Klauzula ta może zostać wykorzystana przykładowo do rozsyłania danych wczytanych z pliku

Dyrektywa omp single

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

int main()
{
    int a;
    int b;

    #pragma omp parallel num_threads(4) private(a,b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;

        #pragma omp single copyprivate(a,b)
        {
            sleep(1);
            cout << "\n";
            a = 20;
            b = 10;
        }

        output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Dyrektywa omp single

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

int main()
{
    int a;
    int b;

    #pragma omp parallel num_threads(4) private(a,b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;

        #pragma omp single copyprivate(a,b)
        {
            sleep(1);
            cout << "\n";
            a = 20;
            b = 10;
        }

        output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy cztery wątki

Dyrektywa omp single

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

int main()
{
    int a;
    int b;

    #pragma omp parallel num_threads(4) private(a,b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;

        #pragma omp single copyprivate(a,b)
        {
            sleep(1);
            cout << "\n";
            a = 20;
            b = 10;
        }

        output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy cztery wątki

Zmienne a oraz b są zmiennymi prywatnymi wątków

Dyrektywa omp single

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

int main()
{
    int a;
    int b;

    #pragma omp parallel num_threads(4) private(a,b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;

        #pragma omp single copyprivate(a,b)
        {
            sleep(1);
            cout << "\n";
            a = 20;
            b = 10;
        }

        output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy cztery wątki

Zmienne a oraz b są zmiennymi prywatnymi wątków

Wartości zmiennych są nieokreślone na początku bloku równoległego

Dyrektywa omp single

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

int main()
{
    int a;
    int b;

    #pragma omp parallel num_threads(4) private(a,b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;

        #pragma omp single copyprivate(a,b)
        {
            sleep(1);
            cout << "\n";
            a = 20;
            b = 10;
        }

        output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy cztery wątki

Zmienne a oraz b są zmiennymi prywatnymi wątków

Wartości zmiennych są nieokreślone na początku bloku równoległego

Każdy wątek wyświetla wartości obu zmiennych

Dyrektywa omp single

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

int main()
{
    int a;
    int b;

    #pragma omp parallel num_threads(4) private(a,b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;

        #pragma omp single copyprivate(a,b)
        {
            sleep(1);
            cout << "\n";
            a = 20;
            b = 10;
        }

        output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy cztery wątki

Zmienne a oraz b są zmiennymi prywatnymi wątków

Wartości zmiennych są nieokreślone na początku bloku równoległego

Każdy wątek wyświetla wartości obu zmiennych

Jeden z wątków modyfikuje wartość swoich zmiennych lokalnych

Dyrektywa omp single

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

int main()
{
    int a;
    int b;

    #pragma omp parallel num_threads(4) private(a,b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;

        #pragma omp single copyprivate(a,b)
        {
            sleep(1);
            cout << "\n";
            a = 20;
            b = 10;
        }

        output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy cztery wątki

Zmienne a oraz b są zmiennymi prywatnymi wątków

Wartości zmiennych są nieokreślone na początku bloku równoległego

Każdy wątek wyświetla wartości obu zmiennych

Po zakończeniu bloku single wartości zmiennych prywatnych a oraz b są rozsyłane do pozostałych wątków

Jeden z wątków modyfikuje wartość swoich zmiennych lokalnych

Dyrektywa omp single

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

int main()
{
    int a;
    int b;

    #pragma omp parallel num_threads(4) private(a,b)
    {
        int threadID = omp_get_thread_num();

        string output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;

        #pragma omp single copyprivate(a,b)
        {
            sleep(1);
            cout << "\n";
            a = 20;
            b = 10;
        }

        output = "Watek #" + to_string(threadID) + ": a=" + to_string(a) + " b=" + to_string(b) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy cztery wątki

Zmienne a oraz b są zmiennymi prywatnymi wątków

Wartości zmiennych są nieokreślone na początku bloku równoległego

Każdy wątek wyświetla wartości obu zmiennych

Po zakończeniu bloku single wartości zmiennych prywatnych a oraz b są rozsyłane do pozostałych wątków

Jeden z wątków modyfikuje wartość swoich zmiennych lokalnych

Ponownie wyświetlamy wartość obu zmiennych

Dyrektywa omp single

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

int main()
{
    int a;
    int b;

    #pragma omp parallel num_thre
    {
        int threadID = omp_get_th

        string output = "Watek #"
        cout << output;

        #pragma omp single copypr
        {
            sleep(1);
            cout << "\n";
            a = 20;
            b = 10;
        }

        output = "Watek #" + to_s
        cout << output;
    }

    return 0;
}
```

Zmienne a oraz b są

Tworzymy c
wątki

lone
go

iennych
łych wątków

rid@gpu2: ~/T3

```
rid@gpu2:~/T3$ g++ -fopenmp t3_omp8.cpp
rid@gpu2:~/T3$ ./a.out
Watek #0: a=1 b=1
Watek #3: a=0 b=0
Watek #1: a=0 b=0
Watek #2: a=0 b=0

Watek #0: a=20 b=10
Watek #2: a=20 b=10
Watek #3: a=20 b=10
Watek #1: a=20 b=10
rid@gpu2:~/T3$
```

Kompilujemy oraz
uruchamiamy program

Wartości zmiennych
przed blokiem `single`

Wartość zmiennych lokalnych
po wykonaniu bloku `single`

Ponownie wyświetlamy
wartość obu zmiennych

Zrównoleglenie obliczeń

- Standard OpenMP dostarcza trzy mechanizmy umożliwiające zrównoleglenie obliczeń:
  - Konstrukcję zrównoleglającą pętlę `for`
  - Zrównoleglenie z wykorzystaniem zadań
  - Mechanizm sekcji

Zrównoleglenie pętli

- Zrównoleglenie pętli jest najczęściej stosowanym sposobem zrównoleglenia programów z wykorzystaniem standardu OpenMP
- W tym celu wykorzystuje się dyrektywę `#pragma omp for`
- Dyrektywa `omp for` występuje wewnątrz obszaru równoległego utworzonego za pomocą dyrektywy `#pragma omp parallel`
  - Obie dyrektywy można połączyć w jedną
- Dyrektywa ta musi występować bezpośrednio przed pętlą `for`
- Z dyrektywą `omp for` nastąpi wspólne (ang. *work sharing*) wykonanie całej pętli – każdy wątek otrzyma część zakresu iteracji

Zrównoleglenie pętli

- Przykład zrównoleglenia pętli z wykorzystaniem dyrektywy `omp for`:

Kod sekwencyjny

```
for(int i=0; i<size-1; ++i)
{
    B[i] = (A[i] + A[i+1]) / 2;
}

for(int i=0; i<size; ++i)
{
    D[i] = 1.0/C[i];
}
```

Kod równoległy

```
#pragma omp parallel num_threads(4)
{
    #pragma omp for
    for(int i=0; i<size-1; ++i)
    {
        B[i] = (A[i] + A[i+1]) / 2;
    }

    #pragma omp for
    for(int i=0; i<size; ++i)
    {
        D[i] = 1.0/C[i];
    }
}
```

Dyrektywa `#pragma omp for` wykorzystana została wewnątrz obszaru równoległego utworzonego wcześniej za pomocą dyrektywy `#pragma omp parallel`

Zrównoleglenie pętli

- Przykład zrównoleglenia pętli z wykorzystaniem dyrektywy `omp for` (wersja skrócona):

Kod sekwencyjny

```
for(int i=0; i<size-1; ++i)
{
    B[i] = (A[i] + A[i+1]) / 2;
}

for(int i=0; i<size; ++i)
{
    D[i] = 1.0/C[i];
}
```

Kod równoległy

```
#pragma omp parallel for num_threads(4)
for(int i=0; i<size-1; ++i)
{
    B[i] = (A[i] + A[i+1]) / 2;
}

#pragma omp parallel for num_threads(4)
for(int i=0; i<size; ++i)
{
    D[i] = 1.0/C[i];
}
```

Dyrektywy `#pragma omp parallel` oraz `#pragma omp for` zostały połączone w pojedynczą dyrektywę

Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[size];

    #pragma omp parallel num_threads(4)
    {
        int threadID = omp_get_thread_num();

        #pragma omp for
        for(int i=0; i<size; ++i)
        {
            array[i] = threadID;
        }
    }

    for(int i=0; i<size; ++i)
    {
        cout << "array[" << i << "] = " << array[i] << endl;
    }

    delete[] array;

    return 0;
}
```

Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[size];

    #pragma omp parallel num_threads(4)
    {
        int threadID = omp_get_thread_num();

        #pragma omp for
        for(int i=0; i<size; ++i)
        {
            array[i] = threadID;
        }
    }

    for(int i=0; i<size; ++i)
    {
        cout << "array[" << i << "] = " << array[i] << endl;
    }

    delete[] array;

    return 0;
}
```

Tworzymy tablicę liczb całkowitych

Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[size];

    #pragma omp parallel num_threads(4)
    {
        int threadID = omp_get_thread_num();

        #pragma omp for
        for(int i=0; i<size; ++i)
        {
            array[i] = threadID;
        }
    }

    for(int i=0; i<size; ++i)
    {
        cout << "array[" << i << "] = " << array[i] << endl;
    }

    delete[] array;

    return 0;
}
```

Tworzymy tablicę liczb całkowitych

Definiujemy obszar równoległy

Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[size];

    #pragma omp parallel num_threads(4)
    {
        int threadID = omp_get_thread_num();

        #pragma omp for
        for(int i=0; i<size; ++i)
        {
            array[i] = threadID;
        }
    }

    for(int i=0; i<size; ++i)
    {
        cout << "array[" << i << "] = " << array[i] << endl;
    }

    delete[] array;

    return 0;
}
```

Tworzymy tablicę liczb całkowitych

Definiujemy obszar równoległy

Region równoległy zawiera cztery wątki

Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[size];

    #pragma omp parallel num_threads(4)
    {
        int threadID = omp_get_thread_num();

        #pragma omp for
        for(int i=0; i<size; ++i)
        {
            array[i] = threadID;
        }
    }

    for(int i=0; i<size; ++i)
    {
        cout << "array[" << i << "] = " << array[i] << endl;
    }

    delete[] array;

    return 0;
}
```

Tworzymy tablicę liczb całkowitych

Definiujemy obszar równoległy

Region równoległy zawiera cztery wątki

Pobieramy ID wątku

Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[size];

    #pragma omp parallel num_threads(4)
    {
        int threadID = omp_get_thread_num();

        #pragma omp for
        for(int i=0; i<size; ++i)
        {
            array[i] = threadID;
        }
    }

    for(int i=0; i<size; ++i)
    {
        cout << "array[" << i << "] = " << array[i] << endl;
    }

    delete[] array;

    return 0;
}
```

Tworzymy tablicę liczb całkowitych

Definiujemy obszar równoległy

Region równoległy zawiera cztery wątki

Pobieramy ID wątku

Wypełniamy tablicę w sposób równoległy:
wątki wpisują swoje ID do elementów tablicy

Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[size];

    #pragma omp parallel num_threads(4)
    {
        int threadID = omp_get_thread_num();

        #pragma omp for
        for(int i=0; i<size; ++i)
        {
            array[i] = threadID;
        }
    }

    for(int i=0; i<size; ++i)
    {
        cout << "array[" << i << "] = " << array[i] << endl;
    }

    delete[] array;

    return 0;
}
```

Tworzymy tablicę liczb całkowitych

Definiujemy obszar równoległy

Region równoległy zawiera cztery wątki

Pobieramy ID wątku

Wypełniamy tablicę w sposób równoległy:
wątki wpisują swoje ID do elementów tablicy

Wyświetlamy zawartość tablicy

Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[s

    #pragma omp parallel n
    {
        int threadID = omp

        #pragma omp for
        for(int i=0; i<siz
        {
            array[i] = thr
        }
    }

    for(int i=0; i<size; +
    {
        cout << "array[" <
    }

    delete[] array;

    return 0;
}
```

Tw

rid@gpu2: ~/T3

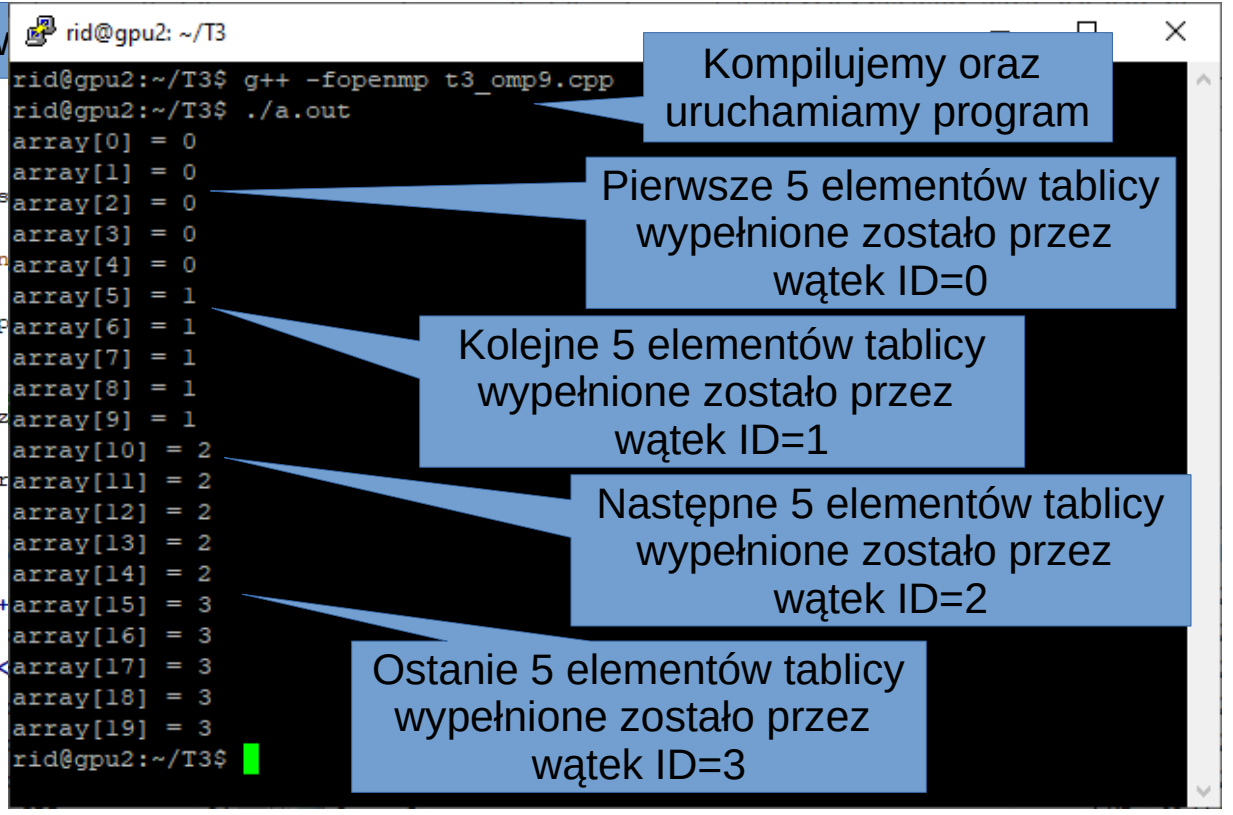
```
rid@gpu2:~/T3$ g++ -fopenmp t3_omp9.cpp
rid@gpu2:~/T3$ ./a.out
array[0] = 0
array[1] = 0
array[2] = 0
array[3] = 0
array[4] = 0
array[5] = 1
array[6] = 1
array[7] = 1
array[8] = 1
array[9] = 1
array[10] = 2
array[11] = 2
array[12] = 2
array[13] = 2
array[14] = 2
array[15] = 3
array[16] = 3
array[17] = 3
array[18] = 3
array[19] = 3
rid@gpu2:~/T3$
```

Kompilujemy oraz uruchamiamy program

Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[s

    #pragma omp parallel n
    {
        int threadID = omp

        #pragma omp for
        for(int i=0; i<siz
        {
            array[i] = thr
        }
    }

    for(int i=0; i<size; +
    {
        cout << "array[" <
    }

    delete[] array;

    return 0;
}
```

Tw



Jak dzielone są iteracje pomiędzy wątki?

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

int main()
{
    const int size = 20;
    int *array = new int[s

    #pragma omp
    {
        int thr

        #pragma
        for(int
        {
            arr
        }
    }

    for(int i=0; i<size; +
    {
        cout << "array[" <
    }

    delete[] array;

    return 0;
}
```

Tw

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp9.cpp
rid@gpu2:~/T3$ ./a.out
array[0] = 0
array[1] = 0
array[14] = 2
array[15] = 3
array[16] = 3
array[17] = 3
array[18] = 3
array[19] = 3
rid@gpu2:~/T3$
```

Kompilujemy oraz uruchamiamy program

Pierwsze 5 elementów tablicy

wątek ID=2

Ostanie 5 elementów tablicy wypełnione zostało przez wątek ID=3

OpenMP dostarcza dedykowaną klauzulę dyrektywy `omp for` umożliwiającą sterowanie sposobem przypisania iteracji pętli do wątków!

Zrównoleglenie za pomocą sekcji

- Nieiteracyjny sposób podziały obliczeń na bloki wykonywane w sposób równoległy
- Każda sekcja wykonywana jest przez jeden wątek
- Sekcje oznaczone są dyrektywą `#pragma omp section` i muszą być umieszczone w obszarze równoległym wewnątrz dyrektywy `#pragma omp sections`
- Może zdarzyć się, że jeden wątek wykona więcej niż jedną sekcję
- Jeżeli w obszarze równoległym utworzone jest więcej wątków niż sekcji, to część z wątków pozostanie bezczynna

Zrównoleglenie za pomocą sekcji

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                suma_elementow(array, size);
            }
            #pragma omp section
            {
                srednia(array, size);
            }
            #pragma omp section
            {
                element_minimalny(array, size);
            }
            #pragma omp section
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Zrównoleglenie za pomocą sekcji

Tworzymy oraz wypełniamy tablicę

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                suma_elementow(array, size);
            }
            #pragma omp section
            {
                srednia(array, size);
            }
            #pragma omp section
            {
                element_minimalny(array, size);
            }
            #pragma omp section
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Zrównoleglenie za pomocą sekcji

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                suma_elementow(array, size);
            }
            #pragma omp section
            {
                srednia(array, size);
            }
            #pragma omp section
            {
                element_minimalny(array, size);
            }
            #pragma omp section
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Definiujemy obszar równoległy zawierający cztery wątki

Zrównoleglenie za pomocą sekcji

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                suma_elementow(array, size);
            }
            #pragma omp section
            {
                srednia(array, size);
            }
            #pragma omp section
            {
                element_minimalny(array, size);
            }
            #pragma omp section
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Definiujemy obszar równoległy zawierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Zrównoleglenie za pomocą sekcji

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                suma_elementow(array, size);
            }
            #pragma omp section
            {
                srednia(array, size);
            }
            #pragma omp section
            {
                element_minimalny(array, size);
            }
            #pragma omp section
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Definiujemy obszar równoległy zawierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Tworzymy cztery sekcje w ramach, których wykonane zostaną różne zadania na danych znajdujących się w tablicy `array`

Zrównoleglenie za pomocą sekcji

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                suma_elementow(array, size);
            }
            #pragma omp section
            {
                srednia(array, size);
            }
            #pragma omp section
            {
                element_minimalny(array, size);
            }
            #pragma omp section
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Definiujemy obszar równoległy zawierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Tworzymy cztery sekcje w ramach, których wykonane zostaną różne zadania na danych znajdujących się w tablicy `array`

Obszar równoległy zawiera cztery wątki, zatem każda z funkcji wykonana zostanie w odrębnym wątku

Zrównoleglenie za pomocą sekcji

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                suma_elementow(array, size);
            }
            #pragma omp section
            {
                srednia(array, size);
            }
            #pragma omp section
            {
                element_minimalny(array, size);
            }
            #pragma omp section
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Definiujemy obszar równoległy zawierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Tworzymy cztery sekcje w ramach, których wykonane zostaną różne zadania na danych znajdujących się w tablicy `array`

Obszar równoległy zawiera cztery wątki, zatem każda z funkcji wykonana zostanie w odrębnym wątku

Na końcu bloku `sections` występuje synchronizacja wątków

Zrównoleglenie za pomocą zadań

- Stadnard OpenMP udostępnia również możliwość podziału pracy za pomocą zadań, które realizowane są asynchronicznie
- Osiągnięcie tego celu możliwe jest z wykorzystaniem dyrektywy `#pragma omp task`
- Konstrukcja ta przypomina sekcje, jednak wykonanie sekcji ograniczone jest do bloku `sections`
- Zadania mogą nie być na stałe przypisane do jednego wątku, a ich wykonanie może zostać odłożone w czasie
- Każdy z wątków posiada prywatną kolejkę, w której umieszczane są zadania. Po jej opróżnieniu sięga po zadania z kolejki innego wątku (ang. *work-stealing*).

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp task
        {
            suma_elementow(array, size);
        }
        #pragma omp task
        {
            srednia(array, size);
        }
        #pragma omp task
        {
            element_minimalny(array, size);
        }
        #pragma omp task
        {
            element_maksymalny(array, size);
        }
    }

    delete[] array;

    return 0;
}
```

Zrównoleglenie za pomocą zadań

Tworzymy oraz wypełniamy tablicę

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp task
        {
            suma_elementow(array, size);
        }
        #pragma omp task
        {
            srednia(array, size);
        }
        #pragma omp task
        {
            element_minimalny(array, size);
        }
        #pragma omp task
        {
            element_maksymalny(array, size);
        }
    }

    delete[] array;

    return 0;
}
```

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp task
        {
            suma_elementow(array, size);
        }
        #pragma omp task
        {
            srednia(array, size);
        }
        #pragma omp task
        {
            element_minimalny(array, size);
        }
        #pragma omp task
        {
            element_maksymalny(array, size);
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp task
        {
            suma_elementow(array, size);
        }
        #pragma omp task
        {
            srednia(array, size);
        }
        #pragma omp task
        {
            element_minimalny(array, size);
        }
        #pragma omp task
        {
            element_maksymalny(array, size);
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp task
        {
            suma_elementow(array, size);
        }
        #pragma omp task
        {
            srednia(array, size);
        }
        #pragma omp task
        {
            element_minimalny(array, size);
        }
        #pragma omp task
        {
            element_maksymalny(array, size);
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Każdy z wątków w obszarze równoległym tworzy cztery zadania z wykorzystaniem dyrektywy `#pragma omp task`

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp task
        {
            suma_elementow(array, size);
        }
        #pragma omp task
        {
            srednia(array, size);
        }
        #pragma omp task
        {
            element_minimalny(array, size);
        }
        #pragma omp task
        {
            element_maksymalny(array, size);
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Każdy z wątków w obszarze równoległym tworzy cztery zadania z wykorzystaniem dyrektywy `#pragma omp task`

W rezultacie, w bloku `parallel` utworzone zostanie 16 zadań

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp task
        {
            suma_elementow(array, size);
        }
        #pragma omp task
        {
            srednia(array, size);
        }
        #pragma omp task
        {
            element_minimalny(array, size);
        }
        #pragma omp task
        {
            element_maksymalny(array, size);
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Każdy z wątków w obszarze równoległym tworzy cztery zadania z wykorzystaniem dyrektywy `#pragma omp task`

W rezultacie, w bloku `parallel` utworzone zostanie 16 zadań

Każda z funkcji wykonana zostanie czterokrotnie

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pra
    {
        }
        #pragma omp task
        {
            element_maksymalny(array, size);
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Jak sprawić, aby każda z funkcji wykonana została tylko raz?

W rezultacie, w bloku `parallel` utworzone zostanie 16 zadań

Każda z funkcji wykonana zostanie czterokrotnie

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                suma_elementow(array, size);
            }
            #pragma omp task
            {
                srednia(array, size);
            }
            #pragma omp task
            {
                element_minimalny(array, size);
            }
            #pragma omp task
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                suma_elementow(array, size);
            }
            #pragma omp task
            {
                srednia(array, size);
            }
            #pragma omp task
            {
                element_minimalny(array, size);
            }
            #pragma omp task
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                suma_elementow(array, size);
            }
            #pragma omp task
            {
                srednia(array, size);
            }
            #pragma omp task
            {
                element_minimalny(array, size);
            }
            #pragma omp task
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Jedynie wątek główny tworzy zadania

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                suma_elementow(array, size);
            }
            #pragma omp task
            {
                srednia(array, size);
            }
            #pragma omp task
            {
                element_minimalny(array, size);
            }
            #pragma omp task
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Jedynie wątek główny tworzy zadania

Utworzone zadania umieszczone zostaną w kolejce zadań wątku głównego

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                suma_elementow(array, size);
            }
            #pragma omp task
            {
                srednia(array, size);
            }
            #pragma omp task
            {
                element_minimalny(array, size);
            }
            #pragma omp task
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Jedynie wątek główny tworzy zadania

Utworzone zadania umieszczone zostaną w kolejce zadań wątku głównego

Pozostałe wątki pobierają zadania z kolejki wątku głównego i je wykonują

Zrównoleglenie za pomocą zadań

```
int main()
{
    int size = 100;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
    {
        array[i] = rand() % 100;
    }

    #pragma omp parallel num_threads(4) shared(array, size)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                suma_elementow(array, size);
            }
            #pragma omp task
            {
                srednia(array, size);
            }
            #pragma omp task
            {
                element_minimalny(array, size);
            }
            #pragma omp task
            {
                element_maksymalny(array, size);
            }
        }
    }

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Tworzymy obszar równoległy zwierający cztery wątki

Zmienne `array` oraz `size` są współdzielone przez wątki

Jedynie wątek główny tworzy zadania

Utworzone zadania umieszczone zostaną w kolejce zadań wątku głównego

Pozostałe wątki pobierają zadania z kolejki wątku głównego i je wykonują

W rezultacie, każda z funkcji wykonana zostanie tylko raz

Synchronizacja wątków

- Synchronizacja wątków stosowana jest do:
  - zabezpieczenia dostępu do danych współdzielonych przez wątki
  - wymuszenia określonej kolejności wykonania obliczeń wewnątrz obszaru równoległego przez poszczególne wątki

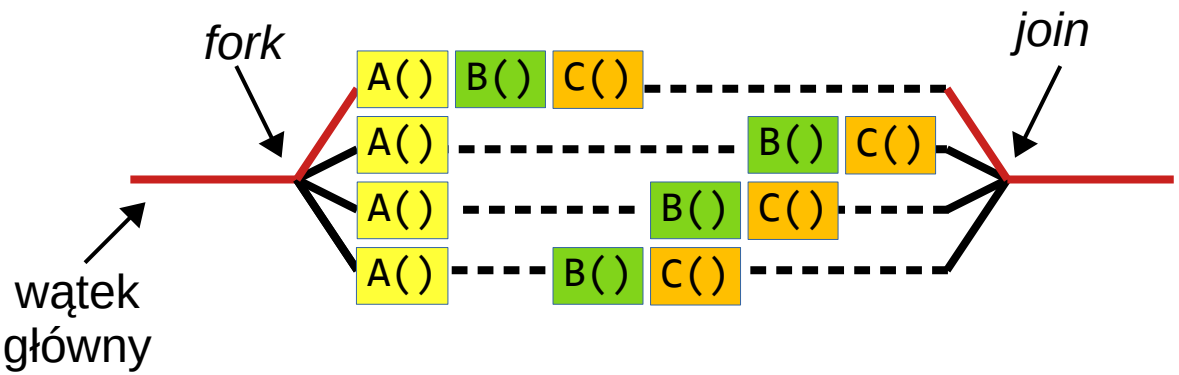
Synchronizacja wątków

- W standardzie OpenMP synchronizację wątków możemy podzielić na:
  - Synchronizację niejawną, która występuje domyślnie na końcu bloku kodu poprzedzonego dyrektywami:
    - `#pragma omp parallel`
    - `#pragma omp for`
    - `#pragma omp single`
    - `#pragma omp sections`
  - Synchronizację jawną definiowaną za pomocą dyrektyw:
    - `#pragma omp critical`
    - `#pragma omp atomic`
    - `#pragma omp barrier`
    - `#pragma omp flush`

Konstrukcja `critical`

- Dyrektywa `#pragma omp critical` tworzy sekcję krytyczną, która może zostać wykonana przez co najwyżej jeden wątek w danym momencie
- Nie można określić kolejności wejścia wątków do sekcji krytycznej
- Służy do unikania sytuacji wyścigu (ang. *race condition*)

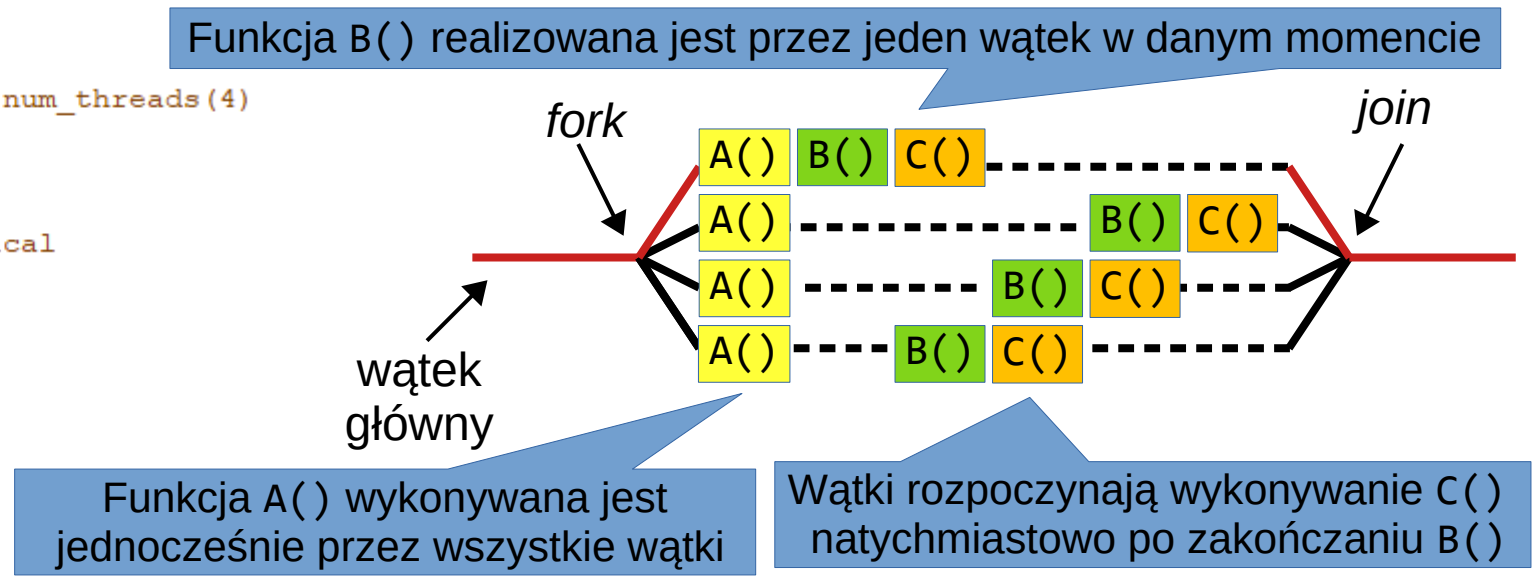
```
#pragma omp parallel num_threads(4)
{
    A();

    #pragma omp critical
    {
        B();
    }

    C();
}
```



Konstrukcja `critical`

- Dyrektywa `#pragma omp critical` tworzy sekcję krytyczną, która może zostać wykonana przez co najwyżej jeden wątek w danym momencie
- Nie można określić kolejności wejścia wątków do sekcji krytycznej
- Służy do unikania sytuacji wyścigu (ang. *race condition*)

```
#pragma omp parallel num_threads(4)
{
    A();

    #pragma omp critical
    {
        B();
    }

    C();
}
```



Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " << avg << endl;
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    average_serial(array, size);
    average_parallel(array, size);

    delete[] array;
    return 0;
}
```

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " << avg << endl;
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    average_serial(array, size);
    average_parallel(array, size);

    delete[] array;
    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " << avg << endl;
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    average_serial(array, size);
    average_parallel(array, size);

    delete[] array;
    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Wyznaczamy średnią z elementów tablicy

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " << avg << endl;
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    average_serial(array, size);
    average_parallel(array, size);

    delete[] array;
    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Wyznaczamy średnią z elementów tablicy

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " << avg << endl;
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    average_serial(array, size);
    average_parallel(array, size);

    delete[] array;
    return 0;
}
```

Kod sekwencyjny wyznaczający średnią

Tworzymy oraz wypełniamy tablicę

Wyznaczamy średnią z elementów tablicy

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " << avg << endl;
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    average_serial(array, size);
    average_parallel(array, size);

    delete[] array;
    return 0;
}
```

Kod sekwencyjny wyznaczający średnią

Kod równoległy wyznaczający średnią

Tworzymy oraz wypełniamy tablicę

Wyznaczamy średnią z elementów tablicy

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " << avg << endl;
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    average_serial(array, size);
    average_parallel(array, size);

    delete[] array;
    return 0;
}
```

Kod sekwencyjny wyznaczający średnią

Kod równoległy wyznaczający średnią

Suma elementów tablicy wyznaczana jest równolegle przez pięć wątków

Tworzymy oraz wypełniamy tablicę

Wyznaczamy średnią z elementów tablicy

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " << avg << endl;
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    average_serial(array, size);
    average_parallel(array, size);

    delete[] array;
    return 0;
}
```

Kod sekwencyjny wyznaczający średnią

Kod równoległy wyznaczający średnią

Suma elementów tablicy wyznaczana jest równolegle przez pięć wątków

Pętla iterująca po indeksach tablicy została zrównoleglona

Tworzymy oraz wypełniamy tablicę

Wyznaczamy średnią z elementów tablicy

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow
}

void average_parallel(const int
{
    int sum = 0;
    #pragma omp parallel num_th
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100

    average_serial(array, size)
    average_parallel(array, size

    delete[] array;
    return 0;
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp10.cpp
rid@gpu2:~/T3$ ./a.out
Srednia elementow tablicy wyznaczona sekwencyjnie: 49
Srednia elementow tablicy wyznaczona rownolegle: 17
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

ą

porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow
}

void average_parallel(const int
{
    int sum = 0;
    #pragma omp parallel num_th
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100

    average_serial(array, size)
    average_parallel(array, size

    delete[] array;
    return 0;
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp10.cpp
rid@gpu2:~/T3$ ./a.out
Srednia elementow tablicy wyznaczona sekwencyjnie: 49
Srednia elementow tablicy wyznaczona rownolegle: 17
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

ą

Wyniki otrzymane dla wersji sekwencyjnej oraz równoległej różnią się

porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow
}

void average_parallel(const int
{
    int sum = 0;
    #pragma omp parallel num_th
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow
}

int main()
{
    srand(time(NULL));
    const int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100

    average_serial(array, size)
    average_parallel(array, size

    delete[] array;
    return 0;
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp10.cpp
rid@gpu2:~/T3$ ./a.out
Srednia elementow tablicy wyznaczona sekwencyjnie: 49
Srednia elementow tablicy wyznaczona rownolegle: 17
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

ą

Wyniki otrzymane dla wersji sekwencyjnej oraz równoległej różnią się

Wniosek?
Wersja równoległa działa niepoprawnie

porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow
}

void average_parallel(const int
{
    int sum = 0;
    #pragma omp parallel num_th
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = sum / n;
    cout << "Srednia elementow
}

int main()
{
    srand(time(NULL))
    const int size =
    int *array = new

    for(int i=0; i<s
        array[i] = ra

    average_serial(a
    average_parallel

    delete[] array;
    return 0;
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp10.cpp
rid@gpu2:~/T3$ ./a.out
Srednia elementow tablicy wyznaczona sekwencyjnie: 49
Srednia elementow tablicy wyznaczona rownolegle: 17
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

ą

Wyniki otrzymane dla wersji sekwencyjnej oraz równoległej różnią się

Wniosek?
Wersja równoległa działa niepoprawnie

Pytanie?
Gdzie jest błąd w kodzie równoległym?

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " << avg << endl;
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            sum += array[i];
    }

    double avg = s
    cout << "Sredn
}

int main()
{
    srand(time(NULL))
    const int size =
    int *array = new

    for(int i=0; i<s
        array[i] = ra

    average_serial(a
    average_parallel

    delete[] array;
    return 0;
}
```

Kod sekwencyjny wyznaczający średnią

Kod równoległy wyznaczający średnią

Suma elementów tablicy wyznaczana jest równolegle przez pięć wątków

Pętla iterująca po indeksach tablicy została zrównoleglona

Wyścig do zmiennej `sum`

Pytanie?
Gdzie jest błąd w kodzie równoległym?

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " <<
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        int sum_local = 0;

        #pragma omp for
        for(int i=0; i<n; ++i)
            sum_local += array[i];

        #pragma omp critical
        {
            sum += sum_local;
        }
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    //...
}
```

Aby rozwiązać problem musimy nieco zmodyfikować wersję równoległą

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " <<
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        int sum_local = 0;

        #pragma omp for
        for(int i=0; i<n; ++i)
            sum_local += array[i];

        #pragma omp critical
        {
            sum += sum_local;
        }
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    //...
}
```

Aby rozwiązać problem musimy nieco zmodyfikować wersję równoległą

Wprowadzamy zmienną prywatną `sum_local`

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " <
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        int sum_local = 0;

        #pragma omp for
        for(int i=0; i<n; ++i)
            sum_local += array[i];

        #pragma omp critical
        {
            sum += sum_local;
        }
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    //...
}
```

Aby rozwiązać problem musimy nieco zmodyfikować wersję równoległą

Wprowadzamy zmienną prywatną `sum_local`

Zmienna ta wykorzystywana jest do sumowania wartości elementów tablicy w obrębie wątku

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona sekwencyjnie: " <
}

void average_parallel(const int *array, const int n)
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(sum)
    {
        int sum_local = 0;

        #pragma omp for
        for(int i=0; i<n; ++i)
            sum_local += array[i];

        #pragma omp critical
        {
            sum += sum_local;
        }
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczona rownolegle: " << avg << endl;
}

int main()
{
    //...
}
```

Aby rozwiązać problem musimy nieco zmodyfikować wersję równoległą

Wprowadzamy zmienną prywatną `sum_local`

Zmienna ta wykorzystywana jest do sumowania wartości elementów tablicy w obrębie wątku

Korzystając z sekcji krytycznej obliczamy ostateczną sumę elementów na podstawie sum lokalnych wątków

Konstrukcja `critical`

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

void average_serial(const int *array, const int n)
{
    int sum = 0;
    for(int i=0; i<n; ++i)
        sum += array[i];

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczo
}

void average_parallel(const int *array, const i
{
    int sum = 0;
    #pragma omp parallel num_threads(5) shared(
    {
        int sum_local = 0;

        #pragma omp for
        for(int i=0; i<n; ++i)
            sum_local += array[i];

        #pragma omp critical
        {
            sum += sum_local;
        }
    }

    double avg = sum / n;
    cout << "Srednia elementow tablicy wyznaczo
}

int main()
{
    //...
}
```

Kor
sum

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_ompl1.cpp
rid@gpu2:~/T3$ ./a.out
Srednia elementow tablicy wyznaczona sekwencyjnie: 51
Srednia elementow tablicy wyznaczona rownolegle: 51
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

Wyniki wersji sekwencyjnej oraz równoległej są jednakowe

Konstrukcja `atomic`

- Specjalnym rodzajem sekcji krytycznej jest operacja atomowa realizowana z wykorzystaniem dyrektywy
  `#pragma omp atomic [klauzula]`
- Dyrektywa ta zapewnia niepodzielność operacji odczytu/zapisu/uaktualnienia określonej komórki pamięci
- Po konstrukcji `atomic` może wystąpić wyrażenie lub blok strukturalny
- Jeżeli architektura procesora, na którym wykonywany jest program wspiera specjalne instrukcje pozwalające na modyfikację fragmentów pamięci w sposób atomowy, realizacja dyrektywy `atomic` będzie wydajniejsze niż dyrektywy `critical`

Konstrukcja `atomic`

- Klauzule konstrukcji `atomic`:
  - `read`: odczytuje wartość zmiennej unikając niebezpieczeństwa odczytu wartości pośredniej zmiennej, gdy jest ona używana jednocześnie przez inny wątek
  - `write`: zapisuje wartość zmiennej, unikając błędów wynikających z równoczesnego zapisu przez wiele wątków
  - `update`: aktualizuje wartość zmiennej. Gwarantuje, że tylko jeden wątek aktualizuje zmienną współdzieloną w danym czasie.
  - `capture`: aktualizuje wartość zmiennej podczas przechwytywania pierwotnej lub końcowej wartości zmiennej
- Jeżeli klauzula dyrektywy `#pragma omp atomic` nie zostanie jawnie zdefiniowana domyślnie wykorzystana zostanie klauzla update

Konstrukcja `atomic`

- Rodzaj operacji atomowej jaki może zostać wykonany uzależniony jest od wykorzystywanej klauzuli:

Klauzula `read`	Klauzula `write`	Klauzula `update`	Klauzula `captured`

Wyrażenie w formie:
`x = v;`
	Wyrażenie w formie:
	`x = expr;`
		Wyrażenie jest jedną z form:
		`x++; x--; ++x; --x;`
		`x binop= expr;`
			Wyrażenie jest jedną z form:
			`x++; x--; ++x; --x; x= v binop= expr;`
			
			lub blok strukturalny jest jedną z form:
			`{v = x; x binop= expr}`
			`{x binop= expr; v = x}`

`x`, `y` – *l-wartości* typu skalarnego

`expr` – wyrażenie typu skalarnego

`binop` – operator binarny +, *, -, /, &, ^, |, << lub >>

`binop`, `binop=`, `++` oraz `--` nie są przeciążone

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Program znajduje w tablicy liczby podzielne bez reszty przez 9

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

Tworzymy oraz wypełniamy tablicę

Program znajduje w tablicy liczby podzielne bez reszty przez 9

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

Wersja sekwencyjna kodu

Tworzymy oraz wypełniamy tablicę

Program znajduje w tablicy liczby podzielne bez reszty przez 9

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

Wersja sekwencyjna kodu

Kod zrównoleglony

Tworzymy oraz wypełniamy tablicę

Program znajduje w tablicy liczby podzielne bez reszty przez 9

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

Wersja sekwencyjna kodu

Kod zrównoleglony

Pętla iterująca po indeksach tablicy została zrównoleglona

Tworzymy oraz wypełniamy tablicę

Program znajduje w tablicy liczby podzielne bez reszty przez 9

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

Wersja sekwencyjna kodu

Kod zrównoleglony

Pętla iterująca po indeksach tablicy została zrównoleglona

W rezultacie, tablica przeszukiwana jest w sposób równoległy

Tworzymy oraz wypełniamy tablicę

Program znajduje w tablicy liczby podzielne bez reszty przez 9

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez re
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

Wersja sekwencyjna kodu

Kod zrównoleglony

Pętla iterująca po indeksach tablicy została zrównoleglona

W rezultacie, tablica przeszukiwana jest w sposób równoległy

Wątki zwiększają licznik `counter`, kiedy wartość danego elementu tablicy jest podzielna przez 9

Tworzymy oraz wypełniamy tablicę

Program znajduje w tablicy liczby podzielne bez reszty przez 9

Aby upewnić się, że wersja równoległa działa poprawnie porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez res
}

void find_parallel(const int *array, const i
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shar
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez re
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp12.cpp
rid@gpu2:~/T3$ ./a.out
Algorytm sekwencyjny
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 92
Algorytm rownolegly
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 43
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

na

równoległy

przez 9

porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez res
}

void find_parallel(const int *array, const i
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shar
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez re
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp12.cpp
rid@gpu2:~/T3$ ./a.out
Algorytm sekwencyjny
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 92
Algorytm rownolegly
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 43
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

Wyniki otrzymane dla wersji sekwencyjnej oraz równoległej różnią się

na

równoległy

przez 9

porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez res
}

void find_parallel(const int *array, const i
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shar
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez re
}

int main()
{
    srand(time(NULL));
    const int number = 9;
    int size = 1000;
    int *array = new int[size];

    for(int i=0; i<size; ++i)
        array[i] = rand() % 100;

    cout << "Algorytm sekwencyjny" << endl;
    find_serial(array, size, number);
    cout << "Algorytm rownolegly" << endl;
    find_parallel(array, size, number);

    delete[] array;

    return 0;
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp12.cpp
rid@gpu2:~/T3$ ./a.out
Algorytm sekwencyjny
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 92
Algorytm rownolegly
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 43
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

Wyniki otrzymane dla wersji sekwencyjnej oraz równoległej różnią się

Wniosek?
Wersja równoległa działa niepoprawnie

przez 9

porównujemy otrzymany wynik z kodem sekwencyjnym

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez res
}

void find_parallel(const int *array, const i
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shar
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez re
}

int main()
{
    srand(time(NULL))
    const int number
    int size = 1000;
    int *array = new

    for(int i=0; i<si
        array[i] = ra

    cout << "Algorytm
    find_serial(array
    cout << "Algorytm
    find_parallel(arr

    delete[] array;

    return 0;
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp12.cpp
rid@gpu2:~/T3$ ./a.out
Algorytm sekwencyjny
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 92
Algorytm rownolegly
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 43
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

Wyniki otrzymane dla wersji sekwencyjnej oraz równoległej różnią się

Wniosek?
Wersja równoległa działa niepoprawnie

Pytanie?
Gdzie jest błąd w kodzie równoległym?

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)

void find_para                    y, const int n, const int number)
{
    int counter =

    #pragma omp paral          hreads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n;  i)
            if(array[i] % number == 0)
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez re
}

int main()
{
    srand(time(NULL))
    const int number
    int size = 1000;
    int *array = new

    for(int i=0; i<si
        array[i] = ra

    cout << "Algorytm
    find_serial(array
    cout << "Algorytm
    find_parallel(arr

    delete[] array;

    return 0;
}
```

Ponownie mamy do czynienia z wyścigiem do zmiennej. Wiele wątków modyfikuje zmienną `counter` jednocześnie

yjna kodu

ter << endl;

Kod zrównoleglony

Pętla iterująca po indeksach tablicy została zrównoleglona

W rezultacie, tablica przeszukiwana jest w sposób równoległy

Wątki zwiększają licznik `counter`, kiedy wartość danego elementu tablicy jest podzielna przez 9

ez 9

Pytanie?
Gdzie jest błąd w kodzie równoległym?

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynos
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                #pragma omp atomic update
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    //...
}
```

Aby rozwiązać problem musimy nieco zmodyfikować wersję równoległą

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                #pragma omp atomic update
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    //...
}
```

Aby rozwiązać problem musimy nieco zmodyfikować wersję równoległą

Aby uniknąć wyścigu do zmiennych wykorzystamy operacje atomowe

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi
}

void find_parallel(const int *array, const int n, const int number)
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(counter)
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                #pragma omp atomic update
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty przez " << number << " wynosi: " << counter << endl;
}

int main()
{
    //...
}
```

Aby rozwiązać problem musimy nieco zmodyfikować wersję równoległą

Aby uniknąć wyścigu do zmiennych wykorzystamy operacje atomowe

Zmienna counter będzie modyfikowana atomowo

Konstrukcja atomic

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

void find_serial(const int *array, const int n, const int number)
{
    int counter = 0;

    for(int i=0; i<n; ++i)
        if(array[i] % number == 0)
            ++counter;

    cout << "Ilosc liczb podzielnych bez reszty
}

void find_parallel(const int *array, const int n
{
    int counter = 0;

    #pragma omp parallel num_threads(5) shared(c
    {
        #pragma omp for
        for(int i=0; i<n; ++i)
            if(array[i] % number == 0)
                #pragma omp atomic update
                ++counter;
    }

    cout << "Ilosc liczb podzielnych bez reszty
}

int main()
{
    //...
}
```

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp13.cpp
rid@gpu2:~/T3$ ./a.out
Algorytm sekwencyjny
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 121
Algorytm rownolegly
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 121
rid@gpu2:~/T3$ ./a.out
Algorytm sekwencyjny
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 101
Algorytm rownolegly
Ilosc liczb podzielnych bez reszty przez 9 wynosi: 101
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

Wyniki wersji sekwencyjnej oraz równoległej są jednakowe

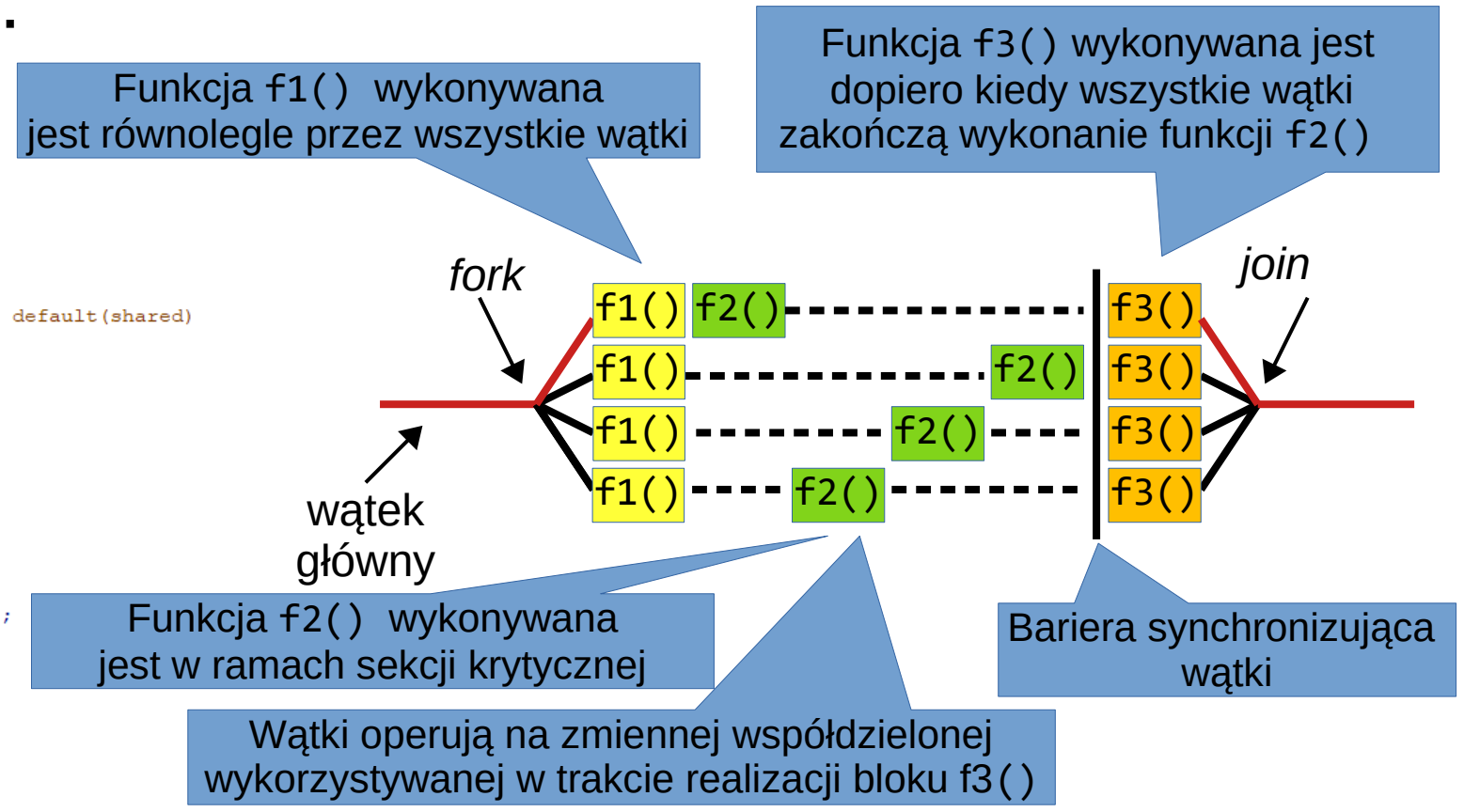
Konstrukcja `barrier`

- Dyrektywa `#pragma omp barrier` wstrzymuje wątki, aż wszystkie wątki zespołu osiągną barierę
- Zapewnia jawny, wymuszony i wspólny dla wszystkich wątków punkt synchronizacji
- Dyrektywa `barrier` występuje niejawnie po konstrukcjach `omp parallel`, `omp for`, `omp sections` oraz `omp single`
- Niejawną synchronizację można usunąć stosując klauzulę `nowait` (działa dla konstrukcji `for`, `sections` oraz `single`)

Konstrukcja `barrier`

- Przykład:

```
int size = 1000;
double arrayA[size];
double arrayB[size];
double arrayC[size];
double ratio = 0.0;

//...

#pragma omp parallel num_threads(4) default(shared)
{
    f1(arrayA, size);

    #pragma omp critial
    {
        ratio += f2(arrayB, size);
    }

    #pragma omp barrier

    f3(arrayC, arrayA, size, ratio);
}

//...
```



Konstrukcja barrier

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

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Konstrukcja barrier

Definiujemy liczbę wątków

```
#include <i
#include <i
#include <o
#include <unistd.h>
using namespace std;

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Konstrukcja barrier

Definiujemy liczbę wątków

Tworzymy tablice o rozmiarze NUM_THREADS

```
#include <i
#include <i
#include <o
#include <unistd.h>
using namespace std;

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Konstrukcja barrier

Definiujemy liczbę wątków

Tworzymy tablice o rozmiarze NUM_THREADS

Tworzymy obszar równoległy zawierający NUM_THREADS wątków

```
#include <i
#include <i
#include <o
#include <unistd.h>
using namespace std;

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Konstrukcja barrier

```
#include <i
#include <i
#include <o
#include <unistd.h>
using namespace std;

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Definiujemy liczbę wątków

Tworzymy tablice o rozmiarze NUM_THREADS

Tworzymy obszar równoległy zawierający NUM_THREADS wątków

Funkcja omp_get_wtime() zwraca liczbę sekund od pewnego ustalonego momentu w przeszłości

Konstrukcja barrier

```
#include <i
#include <i
#include <o
#include <unistd.h>
using namespace std;

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Definiujemy liczbę wątków

Tworzymy tablice o rozmiarze NUM_THREADS

Tworzymy obszar równoległy zawierający NUM_THREADS wątków

Funkcja omp_get_wtime() zwraca liczbę sekund od pewnego ustalonego momentu w przeszłości

Uruchamiamy pomiar czasu

Konstrukcja barrier

```
#include <i
#include <i
#include <o
#include <unistd.h>
using namespace std;

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Definiujemy liczbę wątków

Tworzymy tablice o rozmiarze NUM_THREADS

Tworzymy obszar równoległy zawierający NUM_THREADS wątków

Funkcja omp_get_wtime() zwraca liczbę sekund od pewnego ustalonego momentu w przeszłości

Uruchamiamy pomiar czasu

Usypiamy wątki dwukrotnie na czas delay:
dla wątku ID=0 delay=1, dla wątku ID=1 delay=2, itd...

Konstrukcja barrier

```
#include <i
#include <i
#include <o
#include <unistd.h>
using namespace std;

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Definiujemy liczbę wątków

Tworzymy tablice o rozmiarze `NUM_THREADS`

Tworzymy obszar równoległy zawierający `NUM_THREADS` wątków

Funkcja `omp_get_wtime()` zwraca liczbę sekund od pewnego ustalonego momentu w przeszłości

Uruchamiamy pomiar czasu

Usypiamy wątki dwukrotnie na czas `delay`:
dla wątku ID=0 `delay=1`, dla wątku ID=1 `delay=2`, itd...

Zatrzymujemy pomiar czasu, obliczamy łączny czas uśpienia wątków i zapisujemy rezultat

Konstrukcja barrier

```
#include <i
#include <i
#include <o
#include <unistd.h>
using namespace std;

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Definiujemy liczbę wątków

Tworzymy tablice o rozmiarze `NUM_THREADS`

Tworzymy obszar równoległy zawierający `NUM_THREADS` wątków

Funkcja `omp_get_wtime()` zwraca liczbę sekund od pewnego ustalonego momentu w przeszłości

Uruchamiamy pomiar czasu

Usypiamy wątki dwukrotnie na czas `delay`:
dla wątku ID=0 `delay=1`, dla wątku ID=1 `delay=2`, itd...

Zatrzymujemy pomiar czasu, obliczamy łączny czas uśpienia wątków i zapisujemy rezultat

Wypisujemy czasy uśpienia poszczególnych wątków

Konstrukcja barrier

```
#include <i
#include <i
#include <o
#include <unistd.h>
using namespace std;

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) s
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);


        sleep(dealy);


        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << se

    return 0;
}
```

Definiujemy liczbę wątków

Tworzymy ta

dla wą

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ ./a.out
Watek #0 - czas: 2 sekund
Watek #1 - czas: 4 sekund
Watek #2 - czas: 6 sekund
Watek #3 - czas: 8 sekund
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

Zgodnie z oczekiwaniami każdy wątek usypiany jest na czas wynoszący 2*dealy sekund

Wypisujemy czasy uspienia poszczegolnych wątkow

Konstrukcja barrier

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

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);
        #pragma omp barrier

        sleep(dealy);
        #pragma omp barrier

        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Modyfikujemy nieznacznie kod źródłowy poprzez wprowadzenie dwóch punktów synchronizacji

Konstrukcja barrier

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

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS) shared(threads_times)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);
        #pragma omp barrier

        sleep(dealy);
        #pragma omp barrier

        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << setprecision(3) << threads_times[i] << " sekund" << endl;

    return 0;
}
```

Modyfikujemy nieznacznie kod źródłowy poprzez wprowadzenie dwóch punktów synchronizacji

Dyrektywy `#pragma omp barrier` wywoływane są po każdym uśpieniu wątków

Konstrukcja barrier

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

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);
        #pragma omp barrier

        sleep(dealy);
        #pragma omp barrier

        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << s

    return 0;
}
```

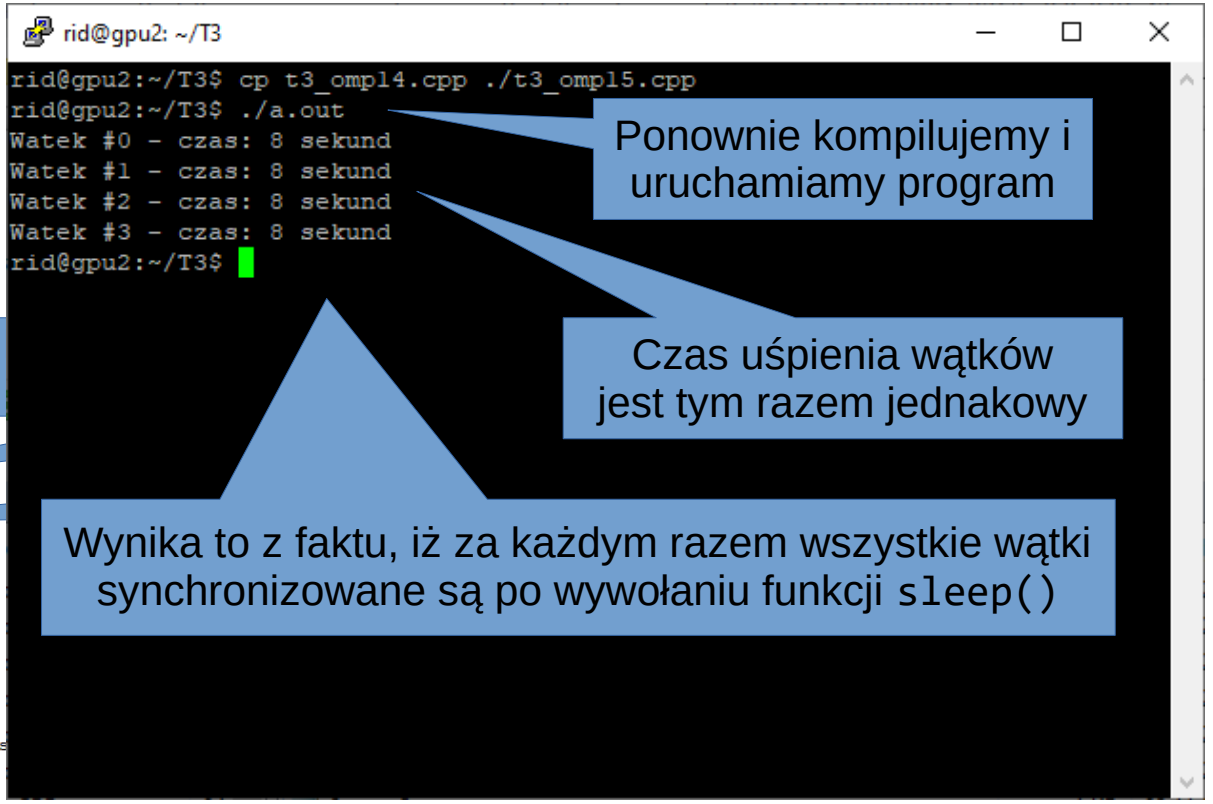
```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ cp t3_omp14.cpp ./t3_omp15.cpp
rid@gpu2:~/T3$ ./a.out
Watek #0 - czas: 8 sekund
Watek #1 - czas: 8 sekund
Watek #2 - czas: 8 sekund
Watek #3 - czas: 8 sekund
rid@gpu2:~/T3$
```

Ponownie kompilujemy i uruchamiamy program

Czas uśpienia wątków jest tym razem jednakowy

Konstrukcja `barrier`

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

#define NUM_THREADS 4

int main()
{
    double threads_times[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; ++i)
        threads_times[i] = 0.0;

    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int threadID = omp_get_thread_num();
        int dealy = threadID + 1;

        double start = omp_get_wtime();

        sleep(dealy);
        #pragma omp barrier

        sleep(dealy);
        #pragma omp barrier

        double stop = omp_get_wtime();
        threads_times[threadID] = (stop - start);
    }

    for(int i=0; i<NUM_THREADS; ++i)
        cout << "Watek #" << i << " - czas: " << s

    return 0;
}
```



Konstrukcja `flush`

- Dyrektywa `#pragma omp flush` może zostać zastosowana do przekazania aktualnej wartości zmiennej pomiędzy wątkami
- Dyrektywa ta określa punkt, w którym wszystkie wątki będą miały ten aktualny i spójny stan pamięci współdzielonej
- Przykładowy scenariusz:
  - Wątek A: zapisuje wartość zmiennej `x`
  - Wątek A: wykonuje dyrektywę `#pragma omp flush` na zmiennej `x`
  - Wątek B: wykonuje dyrektywę `#pragma omp flush` na zmiennej `x`
  - Wątek B: odczytuje **aktualną** wartość zmiennej `x`

Konstrukcja `flush`

```
int main()
{
    //...

    int data;
    int flag = 0;
    #pragma omp parallel sections num_threads(2) shared(data,flag)
    {
        #pragma omp section
        {
            wczytaj(&data);
            #pragma omp flush(data)

            flag = 1;
            #pragma omp flush(flag)

            //...
        }
        #pragma omp section
        {
            while (!flag)
            {
                //...
                #pragma omp flush(flag)
            }

            #pragma omp flush(data)
            oblicz(&data);

            //...
        }
    }

    //...

    return 0;
}
```

Konstrukcja `flush`

Tworzymy obszar równoległy zawierający dwa wątki

```
int main()
{
    //...

    int data;
    int flag = 0;
    #pragma omp parallel sections num_threads(2) shared(data,flag)
    {
        #pragma omp section
        {
            wczytaj(&data);
            #pragma omp flush(data)

            flag = 1;
            #pragma omp flush(flag)

            //...
        }
        #pragma omp section
        {
            while (!flag)
            {
                //...
                #pragma omp flush(flag)
            }

            #pragma omp flush(data)
            oblicz(&data);

            //...
        }
    }

    //...

    return 0;
}
```

Konstrukcja `flush`

```
int main()
{
    //...

    int data;
    int flag = 0;
    #pragma omp parallel sections num_threads(2) shared(data,flag)
    {
        #pragma omp section
        {
            wczytaj(&data);
            #pragma omp flush(data)

            flag = 1;
            #pragma omp flush(flag)

            //...
        }
        #pragma omp section
        {
            while (!flag)
            {
                //...
                #pragma omp flush(flag)
            }

            #pragma omp flush(data)
            oblicz(&data);

            //...
        }
    }

    //...

    return 0;
}
```

Tworzymy obszar równoległy zawierający dwa wątki

Zmienne `data` oraz `flag` są współdzielone przez wątki

Konstrukcja `flush`

```
int main()
{
    //...

    int data;
    int flag = 0;
    #pragma omp parallel sections num_threads(2) shared(data,flag)
    {
        #pragma omp section
        {
            wczytaj(&data);
            #pragma omp flush(data)

            flag = 1;
            #pragma omp flush(flag)

            //...
        }
        #pragma omp section
        {
            while (!flag)
            {
                //...
                #pragma omp flush(flag)
            }

            #pragma omp flush(data)
            oblicz(&data);

            //...
        }
    }

    //...

    return 0;
}
```

Tworzymy obszar równoległy zawierający dwa wątki

Zmienne `data` oraz `flag` są współdzielone przez wątki

Jeden z wątków wczytuje wartość zmiennej `data`

Konstrukcja `flush`

```
int main()
{
    //...

    int data;
    int flag = 0;
    #pragma omp parallel sections num_threads(2) shared(data,flag)
    {
        #pragma omp section
        {
            wczytaj(&data);
            #pragma omp flush(data)

            flag = 1;
            #pragma omp flush(flag)

            //...
        }
        #pragma omp section
        {
            while (!flag)
            {
                //...
                #pragma omp flush(flag)
            }

            #pragma omp flush(data)
            oblicz(&data);

            //...
        }
    }

    //...

    return 0;
}
```

Tworzymy obszar równoległy zawierający dwa wątki

Zmienne `data` oraz `flag` są współdzielone przez wątki

Jeden z wątków wczytuje wartość zmiennej `data`

Następnie dyrektywa `flush` wykonana jest dla zmiennej `data`

Konstrukcja `flush`

```
int main()
{
    //...

    int data;
    int flag = 0;
    #pragma omp parallel sections num_threads(2) shared(data,flag)
    {
        #pragma omp section
        {
            wczytaj(&data);
            #pragma omp flush(data)

            flag = 1;
            #pragma omp flush(flag)

            //...
        }
        #pragma omp section
        {
            while (!flag)
            {
                //...
                #pragma omp flush(flag)
            }

            #pragma omp flush(data)
            oblicz(&data);

            //...
        }
    }

    //...

    return 0;
}
```

Tworzymy obszar równoległy zawierający dwa wątki

Zmienne `data` oraz `flag` są współdzielone przez wątki

Jeden z wątków wczytuje wartość zmiennej `data`

Następnie dyrektywa `flush` wykonana jest dla zmiennej `data`

Te same operacje wykonywane są dla zmiennej `flag`

Konstrukcja `flush`

```
int main()
{
    //...

    int data;
    int flag = 0;
    #pragma omp parallel sections num_threads(2) shared(data,flag)
    {
        #pragma omp section
        {
            wczytaj(&data);
            #pragma omp flush(data)

            flag = 1;
            #pragma omp flush(flag)

            //...
        }
        #pragma omp section
        {
            while (!flag)
            {
                //...
                #pragma omp flush(flag)
            }

            #pragma omp flush(data)
            oblicz(&data);

            //...
        }
    }

    //...

    return 0;
}
```

Tworzymy obszar równoległy zawierający dwa wątki

Zmienne `data` oraz `flag` są współdzielone przez wątki

Jeden z wątków wczytuje wartość zmiennej `data`

Następnie dyrektywa `flush` wykonana jest dla zmiennej `data`

Te same operacje wykonywane są dla zmiennej `flag`

Drugi z wątków wykonuje dyrektywę `flush` i odczytuje aktualną wartość zmiennej `flag`

Konstrukcja `flush`

```
int main()
{
    //...

    int data;
    int flag = 0;
    #pragma omp parallel sections num_threads(2) shared(data,flag)
    {
        #pragma omp section
        {
            wczytaj(&data);
            #pragma omp flush(data)

            flag = 1;
            #pragma omp flush(flag)

            //...
        }
        #pragma omp section
        {
            while (!flag)
            {
                //...
                #pragma omp flush(flag)
            }

            #pragma omp flush(data)
            oblicz(&data);

            //...
        }
    }

    //...

    return 0;
}
```

Tworzymy obszar równoległy zawierający dwa wątki

Zmienne `data` oraz `flag` są współdzielone przez wątki

Jeden z wątków wczytuje wartość zmiennej `data`

Następnie dyrektywa `flush` wykonana jest dla zmiennej `data`

Te same operacje wykonywane są dla zmiennej `flag`

Drugi z wątków wykonuje dyrektywę `flush` i odczytuje aktualną wartość zmiennej `flag`

Po odczytaniu aktualnej wartości zmiennej `flag` wątek kończy obliczenia w pętli `while`

Konstrukcja `flush`

```
int main()
{
    //...

    int data;
    int flag = 0;
    #pragma omp parallel sections num_threads(2) shared(data,flag)
    {
        #pragma omp section
        {
            wczytaj(&data);
            #pragma omp flush(data)

            flag = 1;
            #pragma omp flush(flag)

            //...
        }
        #pragma omp section
        {
            while (!flag)
            {
                //...
                #pragma omp flush(flag)
            }

            #pragma omp flush(data)
            oblicz(&data);

            //...
        }
    }

    //...

    return 0;
}
```

Tworzymy obszar równoległy zawierający dwa wątki

Zmienne `data` oraz `flag` są współdzielone przez wątki

Jeden z wątków wczytuje wartość zmiennej `data`

Następnie dyrektywa `flush` wykonana jest dla zmiennej `data`

Te same operacje wykonywane są dla zmiennej `flag`

Drugi z wątków wykonuje dyrektywę `flush` i odczytuje aktualną wartość zmiennej `flag`

Po odczytaniu aktualnej wartości zmiennej `flag` wątek kończy obliczenia w pętli `while`

Następnie, wątek odczytuje aktualną wartość zmiennej `data` i wykonuje obliczenia

Zagnieżdzone zrównoleglenie

- Standard OpenMP oferuje wsparcie dla zrównoleglenia zagnieżdżonego
- Każdy z wątków w grupie może tworzyć nowe wątki
- Wykorzystanie zagnieżdżonego zrównoleglenia wymaga ustawienia zmiennej środowiskowej `OMP_NESTED` lub wykorzystania funkcji `omp_set_nested()`
- Jeśli zagnieżdżona równoległość nie zostanie włączona, kod będzie wykonywany, ale zespoły wewnętrzne będą zawierać tylko jeden wątek

Zrównoleglenie zagnieżdżone

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

void report_num_threads(const int level)
{
    #pragma omp single
    {
        cout << "Poziom zagniezdzenia: " << level << " - liczba watkow: " << omp_get_num_threads() << endl;
    }
}

int main()
{
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(4)
        {
            report_num_threads(2);
        }
    }

    return 0;
}
```

Zrównoleglenie zagnieżdżone

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

void report_num_threads(const int level)
{
    #pragma omp single
    {
        cout << "Poziom zagniezdzenia: " << level << " - liczba watkow: " << omp_get_num_threads() << endl;
    }
}

int main()
{
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(4)
        {
            report_num_threads(2);
        }
    }

    return 0;
}
```

Włączamy zagnieżdżone zrównoleglenie za pomocą funkcji `omp_set_nested()`

Zrównoleglenie zagnieżdżone

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

void report_num_threads(const int level)
{
    #pragma omp single
    {
        cout << "Poziom zagniezdzenia: " << level << " - liczba watkow: " << omp_get_num_threads() << endl;
    }
}

int main()
{
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(4)
        {
            report_num_threads(2);
        }
    }

    return 0;
}
```

Włączamy zagnieżdżone zrównoleglenie za pomocą funkcji `omp_set_nested()`

Tworzymy dwa wątki

Zrównoleglenie zagnieżdżone

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

void report_num_threads(const int level)
{
    #pragma omp single
    {
        cout << "Poziom zagniezdzenia: " << level << " - liczba watkow: " << omp_get_num_threads() << endl;
    }
}

int main()
{
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(4)
        {
            report_num_threads(2);
        }
    }

    return 0;
}
```

Włączamy zagnieżdżone zrównoleglenie za pomocą funkcji `omp_set_nested()`

Tworzymy dwa wątki

Następnie, każdy z wątków tworzy cztery wątki

Zrównoleglenie zagnieżdżone

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

void report_num_threads(const int level)
{
    #pragma omp single
    {
        cout << "Poziom zagniezdzenia: " << level << " - liczba watkow: " << omp_get_num_threads() << endl;
    }
}

int main()
{
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(4)
        {
            report_num_threads(2);
        }
    }

    return 0;
}
```

Włączamy zagnieżdżone zrównoleglenie za pomocą funkcji `omp_set_nested()`

Tworzymy dwa wątki

Następnie, każdy z wątków tworzy cztery wątki

Wyświetlamy informacje na temat poziomu zagnieżdżenia oraz liczby wątków w zagnieżdżonym obszarze

Zrównoleglenie zagnieżdżone

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

void report_num_threads(const int level)
{
    #pragma omp single
    {
        cout << "Poziom zagniezdzenia: " << level << " - liczba watkow: " << omp_get_num_threads() << endl;
    }
}

int main()
{
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(4)
        {
            report_num_threads(2);
        }
    }

    return 0;
}
```

Jeden z wątków w grupie wyświetla informacje

Włączamy zagnieżdżone zrównoleglenie za pomocą funkcji `omp_set_nested()`

Tworzymy dwa wątki

Następnie, każdy z wątków tworzy cztery wątki

Wyświetlamy informacje na temat poziomu zagnieżdżenia oraz liczby wątków w zagnieżdżonym obszarze

Zrównoleglenie zagnieżdżone

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

void report_num_threads(const int level)
{
    #pragma omp single
    {
        cout << "Poziom zagniezdzenia: " << l
    }
}

int main()
{
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(4)
        {
            report_num_threads(2);
        }
    }

    return 0;
}
```

Jeden z watków w grupie

Włą
za

Wy
poziomu
w zagnieżdżonym obszarze

```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp16.cpp
rid@gpu2:~/T3$ ./a.out
Poziom zagniezdzenia: 1 - liczba watkow: 2
Poziom zagniezdzenia: 2 - liczba watkow: 4
Poziom zagniezdzenia: 2 - liczba watkow: 4
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

Zrównoleglenie zagnieżdżone

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

void report_num_threads(const int level)
{
    #pragma omp single
    {
        cout << "Poziom zagniezdzenia: " << l
    }
}

int main()
{
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(4)
        {
            report_num_threads(2);
        }
    }

    return 0;
}
```

Jeden z watków w grupie

Włą
za

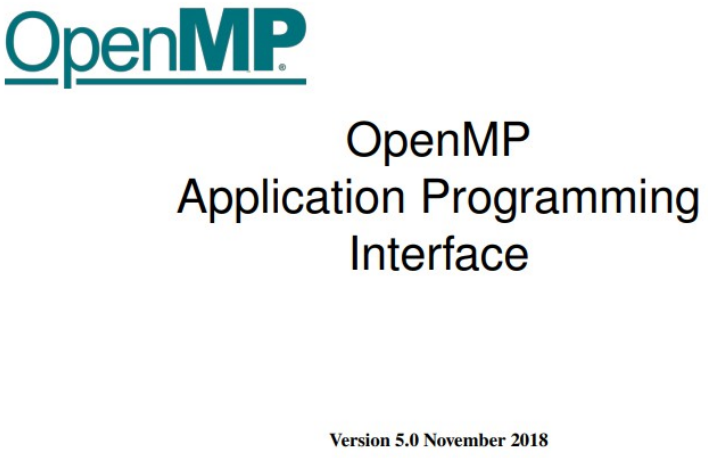
Wy
poziomu
w zagnieżdżonym obszarze

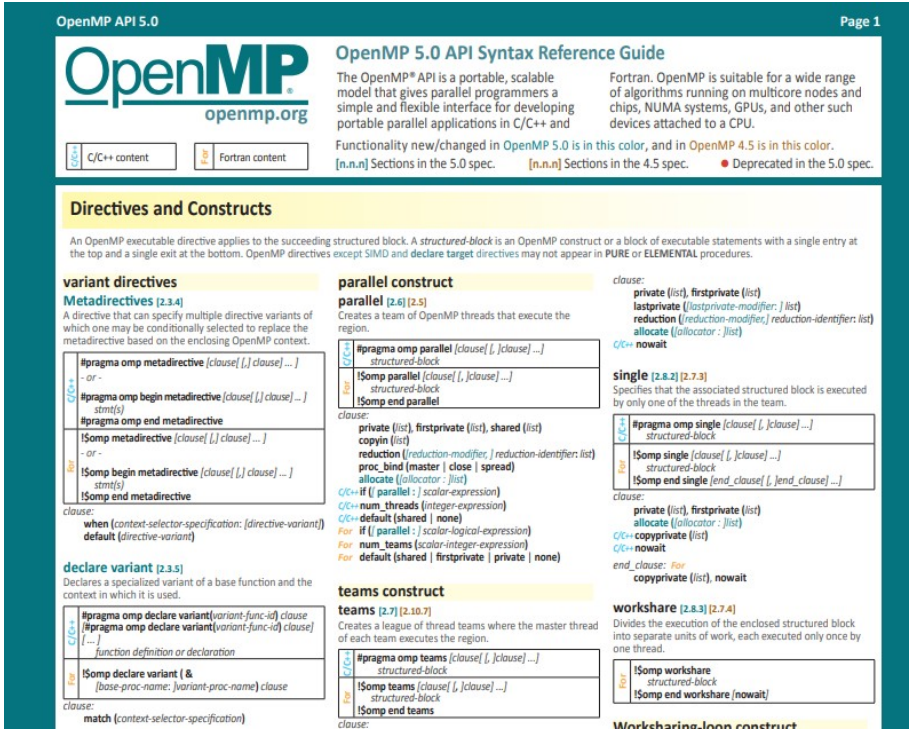
```
rid@gpu2: ~/T3
rid@gpu2:~/T3$ g++ -fopenmp t3_omp16.cpp
rid@gpu2:~/T3$ ./a.out
Poziom zagniezdzenia: 1 - liczba watkow: 2
Poziom zagniezdzenia: 2 - liczba watkow: 4
Poziom zagniezdzenia: 2 - liczba watkow: 4
rid@gpu2:~/T3$
```

Kompilujemy i uruchamiamy program

Uwaga!
Liczba wątków w każdym obszarze zagnieżdżonym może być inna

Literatura





- Dokumentacja standardu OpenMP 5.0: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

Zadanie

Na podstawie zamieszczonych kodów źródłowych sprawdzić we własnym zakresie działanie:

- dyrektywy OpenMP tworzącej wątki;
- metod określania liczby tworzonych wątków;
- klauzul sterujących widocznością zmiennych;
- dyrektywy wykorzystywanej do zrównoleglenia pętli;
- metod serializacji zadań w obszarze równoległym;
- mechanizmów synchronizacji wątków (barierę, sekcję krytyczną, operacje atomowe)

Dziękuję za uwagę!