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 6

Zrównoleglanie programów z wykorzystaniem modelu typu task parallelism

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

Politechnika Częstochowska

Agenda

- Zrównoleglenie programów z wykorzystaniem modelu *task parallelism*, na przykładzie standardu OpenMP:
  - Zrównoleglenie z wykorzystaniem mechanizmu sekcji
  - Zrównoleglenie z wykorzystaniem mechanizmu zadań

Zrównoleglenie z wykorzystaniem mechanizmu sekcji OpenMP

Zrównoleglenie z wykorzystaniem mechanizmu sekcji

- Nieiteracyjny sposób podziału fragmentów programu na bloki (sekcje) wykonywane w sposób równoległy przez wątki programu
- Do zrównoleglenia programu z wykorzystaniem mechanizmu sekcji wykorzystuje się dyrektywę:
  `#pragma omp sections [klauzula1 [klauzula2] …]`
- Lista dostępnych klauzul konstrukcji `sections`: `private`, `firstprivate`, `lastprivate`, `nowait`, `reduction`
- Konstrukcję tą należy umieścić w obszarze równoległym utworzonym za pomocą dyrektywy `#pragma omp parallel`
  - Obie konstrukcje można połączyć w pojedynczą dyrektywę
- Na końcu obszaru poprzedzonego konstrukcją `sections` występuje niejawna synchronizacja wątków (istnieje możliwość jej wyłączenia)

Zrównoleglenie z wykorzystaniem mechanizmu sekcji

- Zadania, które mają zostać wykonane równolegle umieszcza się wewnątrz obszaru oznaczonego konstrukcją `sections`, w blokach poprzedzonych dyrektywą `#pragma omp section`
- Każda sekcja wykonywana jest dokładnie przez jeden wątek
- Poszczególne zadania umieszczone w blokach `section` muszą być od siebie niezależne
- Może zdarzyć się, że jeden wątek wykona więcej niż jedną sekcję, po zakończeniu poprzedniej
- Jeżeli w obszarze równoległym utworzone jest więcej wątków niż sekcji, to część z wątków pozostanie bezczynna

Zrównoleglenie z wykorzystaniem mechanizmu sekcji

- Klauzule dyrektywy `#pragma omp sections`:
  - `private(list)`: zmienne prywatne dla każdego wątku (każdy wątek ma prywatną kopię zmiennych)
  - `firstprivate(list)`: zmienne prywatne dla każdego wątku inicjalizowane wartością zmiennej globalnej
  - `lastprivate(list)`:dane prywatne dla każdego wątku. Wartości są kopiowane na zewnątrz zrównoleglonego rejonu, do zmiennych globalnych o takiej samej nazwie, jeśli bieżąca iteracja jest ostatnią iteracj zrównoleglonej pętli

Zrównoleglenie z wykorzystaniem mechanizmu sekcji

- Klauzule dyrektywy `#pragma omp sections`:
  - `nowait`: wyłącza synchronizację na końcu obszaru. Ustala, że wątek kończący pracę, może rozpocząć kolejne zadanie bez czekania na pozostałe wątki.
  - `reduction(operator | intrinsic: list)`: zmienna posiada lokalną kopię w każdym wątku, lecz wartości lokalnych kopii są podsumowywane (redukowane) do współdzielonej globalnie zmiennej używając zadany operator

Przykład I – Sekcje OpenMP

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

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

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Sekcja 1: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 2: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 3: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Przykład I – Sekcje OpenMP

Tworzymy obszar równoległy

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

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

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Sekcja 1: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 2: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 3: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Przykład I – Sekcje OpenMP

Tworzymy obszar równoległy

Nie określamy jawnie liczby tworzonych wątków

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

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

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Sekcja 1: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 2: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 3: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Przykład I – Sekcje OpenMP

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

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

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Sekcja 1: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 2: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 3: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Tworzymy obszar równoległy

Nie określamy jawnie liczby tworzonych wątków

Pobieramy ID wątków

Przykład I – Sekcje OpenMP

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

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

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Sekcja 1: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 2: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 3: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Tworzymy obszar równoległy

Nie określamy jawnie liczby tworzonych wątków

Pobieramy ID wątków

Zrównoleglamy program z wykorzystaniem sekcji

Przykład I – Sekcje OpenMP

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

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

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Sekcja 1: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 2: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 3: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Tworzymy obszar równoległy

Nie określamy jawnie liczby tworzonych wątków

Pobieramy ID wątków

Zrównoleglamy program z wykorzystaniem sekcji

Tworzymy trzy sekcje

Przykład I – Sekcje OpenMP

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

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

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Sekcja 1: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 2: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 3: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Tworzymy obszar równoległy

Nie określamy jawnie liczby tworzonych wątków

Pobieramy ID wątków

Zrównoleglamy program z wykorzystaniem sekcji

Tworzymy trzy sekcje

W ramach każdej sekcji wyświetlamy informacje o ID wątku, który ją wykonuje

Przykład I – Sekcje OpenMP

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

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

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Sekcja 1: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 2: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                string output = "Sekcja 3: watek #" + to_string(threadID) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Tworzymy obszar równoległy

Nie określamy jawnie liczby tworzonych wątków

Pobieramy ID wątków

Zrównoleglamy program z wykorzystaniem sekcji

Tworzymy trzy sekcje

W ramach każdej sekcji wyświetlamy informacje o ID wątku, który ją wykonuje

Wątki synchronizowane są na końcu obszaru poprzedzonego konstrukcją section

Przykład I – Sekcje OpenMP

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

int mai
{
    #pra
    {

    }

    return 0;
}
```

Tworzymy obszar równoległy

orzonych wątków

staniem sekcji

Tworzymy trzy sekcje

ramach każdej sekcji wyświetlamy
rmacje o ID wątku, który ją wykonuje

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ export OMP_NUM_THREADS=5
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp1.cpp
rid@gpu2:~/RID/T6$ ./a.out
Sekcja 2: watek #3
Sekcja 3: watek #1
Sekcja 1: watek #2
rid@gpu2:~/RID/T6$ ./a.out
Sekcja 1: watek #0
Sekcja 2: watek #1
Sekcja 3: watek #2
rid@gpu2:~/RID/T6$
```

Określamy liczbę wątków
Za pomocną zmiennej
środowiskowej

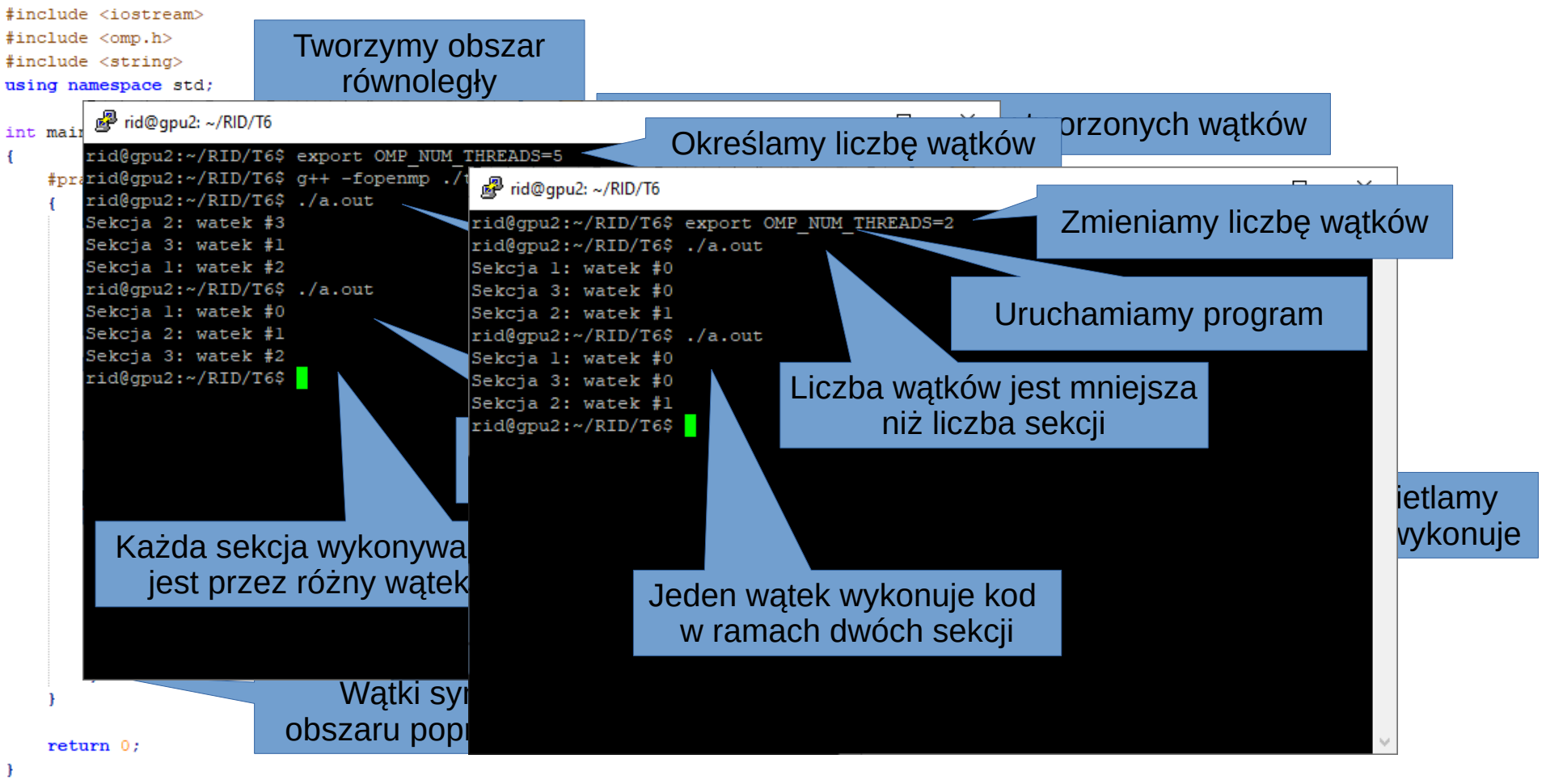
Kompilujemy oraz
uruchamiamy program

Liczba wątków jest większa
niż liczba sekcji

Każda sekcja wykonywana
jest przez różny wątek

Wątki synchronizowane są na końcu
obszaru poprzedzonego konstrukcją section

Przykład I – Sekcje OpenMP



Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy obszar równoległy zawierający 5 wątków

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy obszar równoległy zawierający 5 wątków

Pobieramy ID wątków

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy obszar równoległy zawierający 5 wątków

Pobieramy ID wątków

Pobieramy czas rozpoczęcia realizacji zadań

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy obszar równoległy zawierający 5 wątków

Pobieramy ID wątków

Pobieramy czas rozpoczęcia realizacji zadań

Tworzymy dwie sekcje

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy obszar równoległy zawierający 5 wątków

Pobieramy ID wątków

Pobieramy czas rozpoczęcia realizacji zadań

Tworzymy dwie sekcje

Wątki wykonujące sekcje są usypiane na dwie sekundy

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy obszar równoległy zawierający 5 wątków

Pobieramy ID wątków

Pobieramy czas rozpoczęcia realizacji zadań

Tworzymy dwie sekcje

Wątki wykonujące sekcje są usypiane na dwie sekundy

Po zakończeniu realizacji bloku `sections` wątki są ponownie usypiane na 2 sekundy

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy obszar równoległy zawierający 5 wątków

Pobieramy ID wątków

Pobieramy czas rozpoczęcia realizacji zadań

Tworzymy dwie sekcje

Wątki wykonujące sekcje są usypiane na dwie sekundy

Po zakończeniu realizacji bloku `sections` wątki są ponownie usypiane na 2 sekundy

Pobieramy czas zakończenia realizacji zadań, następnie wyznaczamy łączny czas uśpienia wątków

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Tworzymy obszar równoległy zawierający 5 wątków

Pobieramy ID wątków

Pobieramy czas rozpoczęcia realizacji zadań

Tworzymy dwie sekcje

Wątki wykonujące sekcje są usypiane na dwie sekundy

Po zakończeniu realizacji bloku `sections` wątki są ponownie usypiane na 2 sekundy

Pobieramy czas zakończenia realizacji zadań, następnie wyznaczamy łączny czas uśpienia wątków

Wyświetlamy czas uśpienia wątków

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_
    {
        int threadID = omp_ge

        double start = omp_ge

        #pragma omp sections
        {
            #pragma omp secti
            {
                string output
                cout << outpu
                sleep(2);
            }
            #pragma omp secti
            {
                string output
                cout << outpu
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get
        double sleep_time = s

        string output = "Wate
        cout << output;
    }

    return 0;
}
```

Tworzymy obszar równoległy

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp2.cpp
rid@gpu2:~/RID/T6$ ./a.out
Watek #0: sekcja 1
Watek #3: sekcja 2
Watek #3 - czas uspienia: 4.00
Watek #0 - czas uspienia: 4.00
Watek #1 - czas uspienia: 4.00
Watek #2 - czas uspienia: 4.00
Watek #4 - czas uspienia: 4.00
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Czas uśpienia wszystkich wątków jest jednakowy

Wynika to z faktu, że wątki, które nie wykonują zadań w ramach sekcji są wstrzymywane na końcu bloku `sections`

Wyświetlamy czas uśpienia wątków

ekcje

są usypiane
y

ypiane na 2 sekundy

następnie
ów

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections nowait
        {
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 1" + "\n";
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_string(threadID) + ": sekcja 2" + "\n";
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(threadID) + " - czas uspienia: " + to_string(sleep_time) + "\n";
        cout << output;
    }

    return 0;
}
```

Modyfikujemy program dodając klauzule `nowait` do konstrukcji `sections`

Przykład II – Sekcje OpenMP

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

int main()
{
    #pragma omp parallel num_threads(5)
    {
        int threadID = omp_get_thread_num();

        double start = omp_get_wtime();

        #pragma omp sections nowait
        {
            #pragma omp section
            {
                string output = "Watek #" + to_s
                cout << output;
                sleep(2);
            }
            #pragma omp section
            {
                string output = "Watek #" + to_s
                cout << output;
                sleep(2);
            }
        }

        sleep(2);

        double stop = omp_get_wtime();
        double sleep_time = stop - start;

        string output = "Watek #" + to_string(th
        cout << output;
    }

    return 0;
}
```

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp3.cpp
rid@gpu2:~/RID/T6$ ./a.out
Watek #0: sekcja 1
Watek #3: sekcja 2
Watek #4 - czas uspienia: 2.00
Watek #2 - czas uspienia: 2.00
Watek #1 - czas uspienia: 2.00
Watek #0 - czas uspienia: 4.00
Watek #3 - czas uspienia: 4.00
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Czas uśpienia wątków jest różny

Wątki, które nie wykonują zadań w ramach bloku `sections` nie są wstrzymywane na jego końcu

W rezultacie, ich czas uśpienia jest mniejszy i wynosi jedynie 2 sekundy

Przykład III – Sekcje OpenMP

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

```cpp
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

```cpp
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Funkcja losująca liczby zmiennoprzecinkowe

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Funkcja losująca liczby zmiennoprzecinkowe

Tworzymy trzy tablice

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Funkcja losująca liczby zmiennoprzecinkowe

Tworzymy trzy tablice

Wypełniamy tablice liczbami losowymi

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Funkcja losująca liczby zmiennoprzecinkowe

Tworzymy trzy tablice

Wypełniamy tablice liczbami losowymi

Tworzymy obszar równoległy zwierający trzy wątki

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Funkcja losująca liczby zmiennoprzecinkowe

Tworzymy trzy tablice

Wypełniamy tablice liczbami losowymi

Tworzymy obszar równoległy zwierający trzy wątki

Tworzymy trzy sekcje z wykorzystaniem klauzuli `reduction`

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Funkcja losująca liczby zmiennoprzecinkowe

Tworzymy trzy tablice

Wypełniamy tablice liczbami losowymi

Zmienna `min_avg` jest współdzielona przez wątki

Tworzymy obszar równoległy zwierający trzy wątki

Tworzymy trzy sekcje z wykorzystaniem klauzuli `reduction`

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Funkcja losująca liczby zmiennoprzecinkowe

Tworzymy trzy tablice

Wypełniamy tablice liczbami losowymi

Zmienna `min_avg` jest współdzielona przez wątki

Tworzymy obszar równoległy zwierający trzy wątki

Tworzymy trzy sekcje z wykorzystaniem klauzuli `reduction`

Dla każdej z tablicy wyznaczamy wartość średnią z elementów w osobnej sekcji

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Funkcja losująca liczby zmiennoprzecinkowe

Za pomocą redukcji wyznaczamy wartość maksymalną dla wszystkich wyznaczonych średnich i zapisujemy jej wartość do zmiennej współdzielonej `min_avg`

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Wypełniamy tablice liczbami losowymi

Zmienna jest współdzielona przez wątki

Tworzymy obszar równoległy zwierający trzy wątki

Tworzymy trzy sekcje z wykorzystaniem klauzuli `reduction`

Dla każdej z tablicy wyznaczamy wartość średnią z elementów w osobnej sekcji

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Funkcja losująca liczby zmiennoprzecinkowe

Za pomocą redukcji wyznaczamy wartość maksymalną dla wszystkich wyznaczonych średnich i zapisujemy jej wartość do zmiennej współdzielonej `min_avg`

Tworzymy trzy tablice

Wypełniamy tablice liczbami losowymi

Zmienna jest współdzielona przez wątki

Tworzymy obszar równoległy zwierający trzy wątki

Tworzymy trzy sekcje z wykorzystaniem klauzuli `reduction`

Dla każdej z tablicy wyznaczamy wartość średnią z elementów w osobnej sekcji

Wyświetlamy największą wartość średnią

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;

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

        #pragma omp sections reduction(max : max_avg)
        {
            #pragma omp section
            {
                max_avg = average(A, size);
                string output = "Sekcja #1: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(B, size);
                string output = "Sekcja #2: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
            #pragma omp section
            {
                max_avg = average(C, size);
                string output = "Sekcja #3: Watek #" + to_string(threadID) + " srednia = " + to_string(max_avg) + "\n";
                cout << output;
            }
        }
    }
    cout << "Max avg. " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład III – Sekcje OpenMP

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND
    return f + (double)(rand() % 100
}
```

Funkcja losująca liczby zmiennoprzecinkowe

Za pomocą redukcji wyz
wartość maksymalną dla
wyznaczonych średn
zapisujemy jej wartość do zmiennej współdzielonej `min_avg`

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];

    for(int i=0; i<size; ++i)
    {
```

Tworzymy trzy tablice

Wypełniamy tablice liczbami losowymi

```
    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Wyświetlamy największą wartość średnią

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp4.cpp
rid@gpu2:~/RID/T6$ ./a.out
Sekcja #2: Watek #0 srednia = 482.134122
Sekcja #1: Watek #1 srednia = 541.569775
Sekcja #3: Watek #2 srednia = 522.220817
Max avg. 541.57
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

W ramach każdej sekcji wyznaczana jest wartość średnia dla innej tablicy

Wartość maksymalna ze wszystkich średnich wyznaczona za pomocą redukcji

ółdzielona przez wątki

noległy zwierający ątki

e z wykorzystaniem eduction

(max_avg) + "\n";

(max_avg) + "\n";

(max_avg) + "\n";

wyznaczamy wartość tów w osobnej sekcji

Przykład IV – Sekcje OpenMP

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

```cpp
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Przykład IV – Sekcje OpenMP

Wyznaczamy wartość liczby Pi trzema metodami

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

```cpp
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

```cpp
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

Metoda Willisa

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \ldots = \frac{PI}{2}$$

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

```
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

Metoda Willisa

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \ldots = \frac{PI}{2}$$

Metoda Leibnitza

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \ldots = \frac{PI}{4}$$

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

```
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

Metoda Willisa

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \ldots = \frac{PI}{2}$$

Metoda Leibnitza

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \ldots = \frac{PI}{4}$$

Wartość Pi wyznaczamy poprzez obliczenie wartości całki oznaczonej

$$\int_0^1 \frac{4}{1+x^2} dx = PI$$

```
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

Metoda Willisa

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \ldots = \frac{PI}{2}$$

Metoda Leibnitza

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \ldots = \frac{PI}{4}$$

Wartość Pi wyznaczamy poprzez obliczenie wartości całki oznaczonej

$$\int_0^1 \frac{4}{1+x^2} dx = PI$$

Wszystkie metody wykonywane są sekwencyjnie

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

```
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

Metoda Willisa

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \ldots = \frac{PI}{2}$$

Metoda Leibnitza

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \ldots = \frac{PI}{4}$$

Wartość Pi wyznaczamy poprzez obliczenie wartości całki oznaczonej

$$\int_0^1 \frac{4}{1+x^2} dx = PI$$

```
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Wszystkie metody wykonywane są sekwencyjnie

Kod równoległy: wszystkie metody wykonywane są jednocześnie

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

Metoda Willisa

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \ldots = \frac{PI}{2}$$

Metoda Leibnitza

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \ldots = \frac{PI}{4}$$

Wartość Pi wyznaczamy poprzez obliczenie wartości całki oznaczonej

$$\int_0^1 \frac{4}{1+x^2} dx = PI$$

Wszystkie metody wykonywane są sekwencyjnie

Tworzymy obszar równoległy z trzema wątkami

Kod równoległy: wszystkie metody wykonywane są jednocześnie

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

```
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

Metoda Willisa

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \ldots = \frac{PI}{2}$$

Metoda Leibnitza

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \ldots = \frac{PI}{4}$$

Wartość Pi wyznaczamy poprzez obliczenie wartości całki oznaczonej

$$\int_0^1 \frac{4}{1+x^2} dx = PI$$

```
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Wszystkie metody wykonywane są sekwencyjnie

Tworzymy obszar równoległy z trzema wątkami

Tworzymy trzy sekcje

Kod równoległy: wszystkie metody wykonywane są jednocześnie

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

Metoda Willisa

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \ldots = \frac{PI}{2}$$

Metoda Leibnitza

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \ldots = \frac{PI}{4}$$

Wartość Pi wyznaczamy poprzez obliczenie wartości całki oznaczonej

$$\int_0^1 \frac{4}{1+x^2} dx = PI$$

```
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Wszystkie metody wykonywane są sekwencyjnie

Tworzymy obszar równoległy z trzema wątkami

Dla obu wersji mierzymy całkowity czas obliczeń

Tworzymy trzy sekcje

Kod równoległy: wszystkie metody wykonywane są jednocześnie

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1.0));
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0));
        else
            ai = (double)(-1.0 / (2.0*i + 1.0));
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

Metoda Willisa

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \ldots = \frac{PI}{2}$$

Metoda Leibnitza

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \ldots = \frac{PI}{4}$$

Wartość Pi wyznaczamy poprzez obliczenie wartości całki oznaczonej

$$\int_0^1 \frac{4}{1+x^2} dx = PI$$

```
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double stop_serial = omp_get_wtime();
    double time_serial = (stop_serial-start_serial);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_s << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_s << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_s << endl;
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s] \n";

    cout << "\n";
    cout << "Wersja rownolegla..." << "\n";
    double pi1_p, pi2_p, pi3_p;
    double start_parallel = omp_get_wtime();
    #pragma omp parallel num_threads(3)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                pi1_p = pi_wallis(steps);
            }
            #pragma omp section
            {
                pi2_p = pi_leibnitz(steps);
            }
            #pragma omp section
            {
                pi3_p = pi_calkowanie(steps);
            }
        }
    }
    double stop_parallel = omp_get_wtime();
    double time_parallel = (stop_parallel-start_parallel);
    cout << "   PI metoda Wallisa: " << setprecision(15) << pi1_p << endl;
    cout << "   PI metoda Leibnitza: " << setprecision(15) << pi2_p << endl;
    cout << "   PI poprzez calkowanie: " << setprecision(15) <<  pi3_p << endl;
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s] \n";
    cout << "\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

Wszystkie metody wykonywane są sekwencyjnie

Tworzymy obszar równoległy z trzema wątkami

Dla obu wersji mierzymy całkowity czas obliczeń

Tworzymy trzy sekcje

Kod równoległy: wszystkie metody wykonywane są jednocześnie

Wyznaczamy przyspieszenie

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0
        else
            ai = (double)(-1.0 / (2.0*i + 1.
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

```cpp
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
    pi2_s = pi_leibnitz(steps);
    pi3_s = pi_calkowanie(steps);
    double time_serial = omp_get_wtime() - start_serial;

    cout << "   PI metoda Wallisa: " << setprecision(14) << pi1_s << "\n";
    cout << "   PI metoda Leibnitza: " << setprecision(14) << pi2_s << "\n";
    cout << "   PI poprzez calkowanie: " << setprecision(14) << pi3_s << "\n";
    cout << "Czas obliczen wykonywanych sekwencyjnie: " << time_serial << " [s]\n\n";

    double pi1_p, pi2_p, pi3_p;

    double start_parallel = omp_get_wtime();
    #pragma omp parallel sections
    {
        #pragma omp section
        pi1_p = pi_wallis(steps);
        #pragma omp section
        pi2_p = pi_leibnitz(steps);
        #pragma omp section
        pi3_p = pi_calkowanie(steps);
    }
    double time_parallel = omp_get_wtime() - start_parallel;

    cout << "Wersja rownolegla..." << "\n";
    cout << "   PI metoda Wallisa: " << setprecision(14) << pi1_p << "\n";
    cout << "   PI metoda Leibnitza: " << setprecision(14) << pi2_p << "\n";
    cout << "   PI poprzez calkowanie: " << setprecision(14) << pi3_p << "\n";
    cout << "Czas obliczen wykonywanych rownolegle: " << time_parallel << " [s]\n\n";

    cout << "Przyspiesznie: " << setprecision(4) << time_serial / time_parallel << "\n";
    return 0;
}
```

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp5.cpp
rid@gpu2:~/RID/T6$ ./a.out
Wersja sekwencyjna...
   PI metoda Wallisa: 3.14159264306626
   PI metoda Leibnitza: 3.14159265158926
   PI poprzez calkowanie: 3.14159265358981
Czas obliczen wykonywanych sekwencyjnie: 11.0768672120757 [s]

Wersja rownolegla...
   PI metoda Wallisa: 3.14159264306626
   PI metoda Leibnitza: 3.14159265158926
   PI poprzez calkowanie: 3.14159265358981
Czas obliczen wykonywanych rownolegle: 5.38093229895458 [s]

Przyspiesznie: 2.059
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Wszystkie metody wykonywane są sekwencyjnie

Wyniki oraz czas obliczeń wersji sekwencyjnej

Dla obu wersji mierzymy całkowity czas obliczeń

Wyniki oraz czas obliczeń wersji równoległej

Kod równoległy: wszystkie metody wykonywane są jednocześnie

Obliczenia równoległe wykonywane są ponad 2 razy szybciej

Wyznaczamy przyspieszenie

Przykład IV – Sekcje OpenMP

Parametr `steps` określa dokładność przybliżenia liczby Pi

Wyznaczamy wartość liczby Pi trzema metodami

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

double pi_wallis(const int steps)
{
    double pi, ai;
    double tmp = 1.0;
    for (long int i=1; i<steps; ++i)
    {
        ai = (double)(4.0*i*i / (4.0*i*i - 1
        tmp = tmp * ai;
    }

    pi = (double)(2.0 * tmp);
    return pi;
}

double pi_leibnitz(const int steps)
{
    double pi, ai;
    double tmp = 0.0;
    for (long int i=0; i<steps; ++i)
    {
        if (i % 2 == 0)
            ai = (double)(1.0 / (2.0*i + 1.0
        else
            ai = (double)(-1.0 / (2.0*i + 1.
        tmp = tmp + ai;
    }

    pi = (double)(4.0 * tmp);
    return pi;
}

double pi_calkowanie(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}
```

```cpp
int main()
{
    const int steps = 500000000;
    cout << "Wersja sekwencyjna..." << "\n";
    double pi1_s, pi2_s, pi3_s;

    double start_serial = omp_get_wtime();
    pi1_s = pi_wallis(steps);
```

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp5.cpp
rid@gpu2:~/RID/T6$ ./a.out
Wersja sekwencyjna...
   PI metoda Wallisa: 3.14159264306626
   PI metoda Leibnitza: 3.14159265158926
   PI poprzez calkowanie: 3.14159265358981
Czas obliczen wykonywanych sekwencyjnie: 11.0768672120757 [s]

Wersja rownolegla...
   PI metoda Wallisa: 3.14159264306626
   PI metoda Leibnitza: 3.14159265158926
   PI poprzez calkowanie: 3.14159265358981
Czas obliczen wykonywanych rownolegle: 5.38093229895458 [s]

Przyspiesznie: 2.059
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Wszystkie metody wykonywane są sekwencyjnie

Dla obu wersji mierzymy całkowity czas obliczeń

Wyniki oraz czas obliczeń wersji sekwencyjnej

Wyniki oraz czas obliczeń wersji równoległej

Kod równoległy: wszystkie metody wykonywane są

Obliczenia równoległe wykonywane są ponad 2 razy szybciej

Wszystkie metody realizowane są jednocześnie, zatem czas wykonania obliczeń równoległych odpowiada czasowi obliczeń metody, która wykonana jest najdłużej

przyspieszenie

Zrównoleglenie z wykorzystaniem mechanizmu zadań OpenMP

Zrównoleglenie za pomocą zadań

- Stadnard OpenMP udostępnia również możliwość podziału pracy za pomocą zadań
- Osiągnięcie tego celu możliwe jest z wykorzystaniem następującej dyrektywy:
  `#pragma omp task [klauzula1 [klauzula2]…]`
- Konstrukcja ta przypomina sekcje, jednak wykonanie sekcji ograniczone jest do bloku `sections`
- Zadania są kolejkowanie i wykonywane we wskazanych punktach synchronizacji
- Jeżeli nie ma jawnie określonych punktów synchronizacji, to zadania mogą zostać wykonane w dowolnym momencie wewnątrz bloku `parallel`
- Każdy z wątków posiada prywatną kolejkę, w której umieszczane są zadania. Po jej opróżnieniu, sięga po zadania z kolejki innego wątku (ang. *work-stealing*).
- Zadania mogą nie być na stałe przypisane do jednego wątku, a ich wykonanie może zostać odłożone w czasie

Zrównoleglenie za pomocą zadań

- Klauzule dyrektywy `#pragma omp task`:
  - `default(shared|none)`: definiuje domyślny zakres widoczności zmiennych – wszystkie (`shared`) lub żadne (`none`) zmienne nie są współdzielone w ramach zadania
  - `private(list)`: zmienne na liście traktowane są jako prywatne w ramach zadania
  - `firstprivate(list)`: zmienne na liście należy uznać za prywatne, które inicjalizowane są wartością zmiennej globalnej
  - `shared(list)`: określa listę zmiennych współdzielonych wykorzystywanych w ramach zadania

Zrównoleglenie za pomocą zadań

- Klauzule dyrektywy `#pragma omp task`:
  - `if`(wyrażenie): jeżeli wartość wyrażenia jest nieprawdziwa, kod zadania wykonany zostanie natychmiastowo bez generowania instancji zadania (zadanie „nieodroczone”)
  - `untied`: określa, że po zawieszeniu wykonania, zadanie nie zostanie wykonane przez wątek, który je utworzył. Domyślnie zadania są `tied` (zawsze wykonywane przez wątek, który je utworzył).
  - `final(wyrazenie)`: jeśli wyrażenie jest prawdziwe zadania potomne wykonane zostaną od razu (zadania potomne są „wstawione”)
  - `mergable`: klauzula dla zadań zagnieżdzonych. Pozwala łączyć zadania jeśli są „nieodroczone” lub „wstawione”. Zadania scalone mogą korzystać ze środowiska danych zadania nadrzędnego (rodzica).

Zrównoleglenie za pomocą zadań

- Klauzule dyrektywy `#pragma omp task`:
  - `depend(rodzaj_zaleznosci:lista)`: określa zależności między zadaniami. Pozwala na zdefiniowanie kolejności wykonywania zadań.
    - Trzy rodzaje zależności: `in`, `out`, `inout`
    - `in`: wygenerowane zadanie będzie zależne od wcześniej utworzonych zadań, które odwołują się do zmiennych wyszczególnionych na liście za pomocą klauzul `out` lub `inout`
    - `out` oraz `inout`: wygenerowane zadanie będzie zależne od wcześniej utworzonych zadań, które odwołują się do zmiennych wyszczególnionych na liście za pomocą klauzul `in`, `out` lub `inout`
    - Elementy listy w klauzuli `depend` mogą zawierać sekcje tablicowe

Zrównoleglenie za pomocą zadań

- Klauzule dyrektywy `#pragma omp task`:
  - `affinity(lista)`: przydatna dla obliczeń realizowanych w systemach wieloprocesorowych o architekturze NUMA. Zaleca się wykonanie zadania w pobliżu fizycznej pozycji elementów na liście.
  - `in_reduction(operator | intrinsic: list)`: rozszerza tradycyjną redukcję używaną przykładowo dla dyrektywy `omp for`. Umożliwia zastosowanie redukcji dla grafów zadań.
  - `priority(wartosc)`: określa priorytet zadania. Przyjmuje wartości nieujemne. Zadania o wyższym priorytecie są wykonywane przed zadaniami o niskim priorytecie.

Przykład I – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task
        {
            taskA();
        }

        #pragma omp task
        {
            taskB();
        }

        #pragma omp task
        {
            taskC();
        }
    }

    return 0;
}
```

Przykład I – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task
        {
            taskA();
        }

        #pragma omp task
        {
            taskB();
        }

        #pragma omp task
        {
            taskC();
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Przykład I – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task
        {
            taskA();
        }

        #pragma omp task
        {
            taskB();
        }

        #pragma omp task
        {
            taskC();
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Przykład I – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task
        {
            taskA();
        }

        #pragma omp task
        {
            taskB();
        }

        #pragma omp task
        {
            taskC();
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Tworzymy obszar równoległy zawierający trzy wątki

Przykład I – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task
        {
            taskA();
        }

        #pragma omp task
        {
            taskB();
        }

        #pragma omp task
        {
            taskC();
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Tworzymy obszar równoległy zawierający trzy wątki

Każdy z wątków tworzy trzy zadania

Przykład I – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task
        {
            taskA();
        }

        #pragma omp task
        {
            taskB();
        }

        #pragma omp task
        {
            taskC();
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Tworzymy obszar równoległy zawierający trzy wątki

Każdy z wątków tworzy trzy zadania

Każde zadanie opowiada za wykonanie innej funkcji

Przykład I – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task
        {
            taskA();
        }

        #pragma omp task
        {
            taskB();
        }

        #pragma omp task
        {
            taskC();
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Tworzymy obszar równoległy zawierający trzy wątki

Każdy z wątków tworzy trzy zadania

Każde zadanie opowiada za wykonanie innej funkcji

W rezultacie utworzone zostanie dziewięć zadań

Przykład I – Tworzenie zadań

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

void taskA()
{
    string output = "Hello w
    cout << output;
}

void taskB()
{
    string output = "Hello w
    cout << output;
}

void taskC()
{
    string output = "Hello w
    cout << output;
}

int main()
{
    #pragma omp parallel num
    {
        #pragma omp task
        {
            taskA();
        }

        #pragma omp task
        {
            taskB();
        }

        #pragma omp task
        {
            taskC();
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp6.cpp
rid@gpu2:~/RID/T6$ ./a.out
Hello world from Task A
Hello world from Task B
Hello world from Task A
Hello world from Task B
Hello world from Task A
Hello world from Task C
Hello world from Task C
Hello world from Task B
Hello world from Task C
rid@gpu2:~/RID/T6$
```

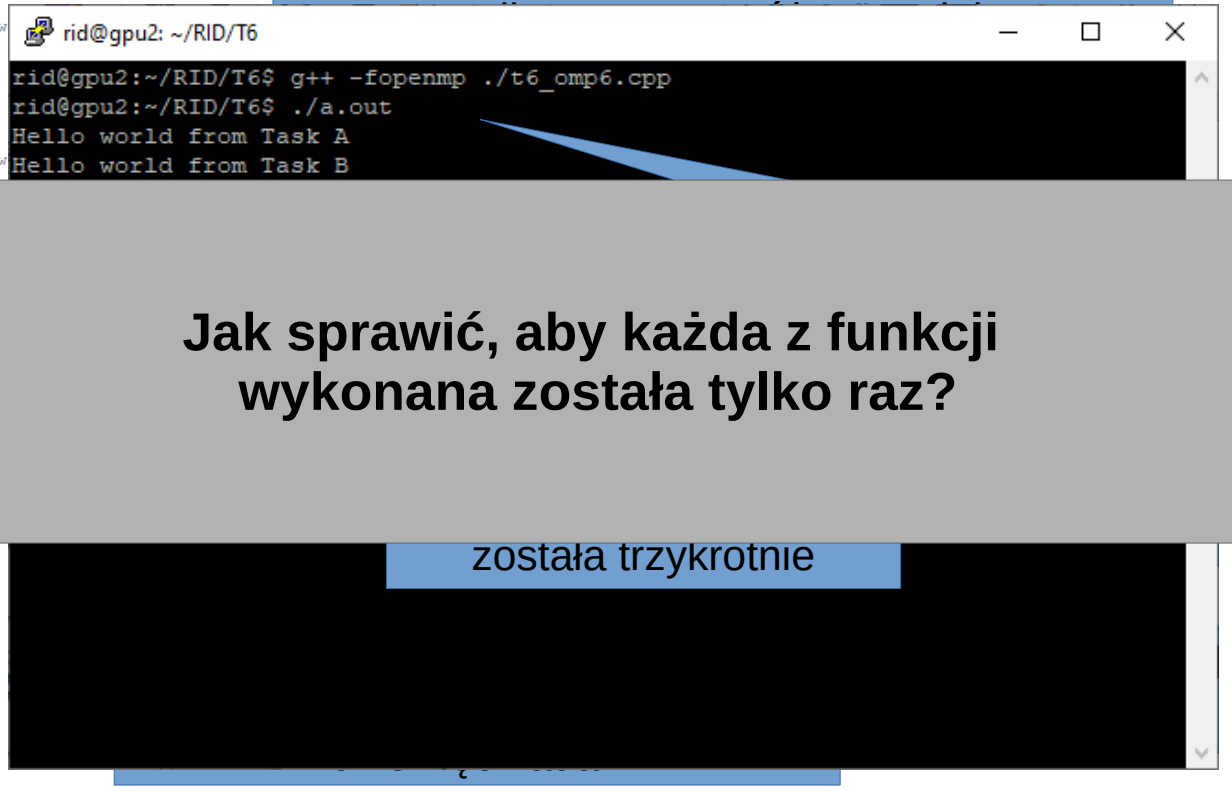
Kompilujemy oraz uruchamiamy program

Każda funkcja wykonana została trzykrotnie

Przykład I – Tworzenie zadań

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

void taskA()
{
    string output = "Hello w
    cout << output;
}

void taskB()
{
    string output = "Hello w
    cout <<
}

void taskC(
{
    string
    cout <<
}

int main()
{
    #pragma
    {
        #pr
        {
        }

        #pragma omp task
        {
            taskB();
        }

        #pragma omp task
        {
            taskC();
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których



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

została trzykrotnie

Przykład II – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                taskA();
            }

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Przykład II – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                taskA();
            }

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Przykład II – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                taskA();
            }

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Tworzymy obszar równoległy zawierający trzy wątki

Przykład II – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                taskA();
            }

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Tworzymy obszar równoległy zawierający trzy wątki

Wątek główny tworzy trzy zadania i umieszcza jej w swojej prywatnej kolejce zadań

Przykład II – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                taskA();
            }

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Tworzymy obszar równoległy zawierający trzy wątki

Wątek główny tworzy trzy zadania i umieszcza jej w swojej prywatnej kolejce zadań

Każdy wątek pobiera zadanie z kolejki zadań wątku głównego i je wykonuje

Przykład II – Tworzenie zadań

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

void taskA()
{
    string output = "Hello world from Task A\n";
    cout << output;
}

void taskB()
{
    string output = "Hello world from Task B\n";
    cout << output;
}

void taskC()
{
    string output = "Hello world from Task C\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task
            {
                taskA();
            }

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla prosty komunikat

Tworzymy obszar równoległy zawierający trzy wątki

Wątek główny tworzy trzy zadania i umieszcza jej w swojej prywatnej kolejce zadań

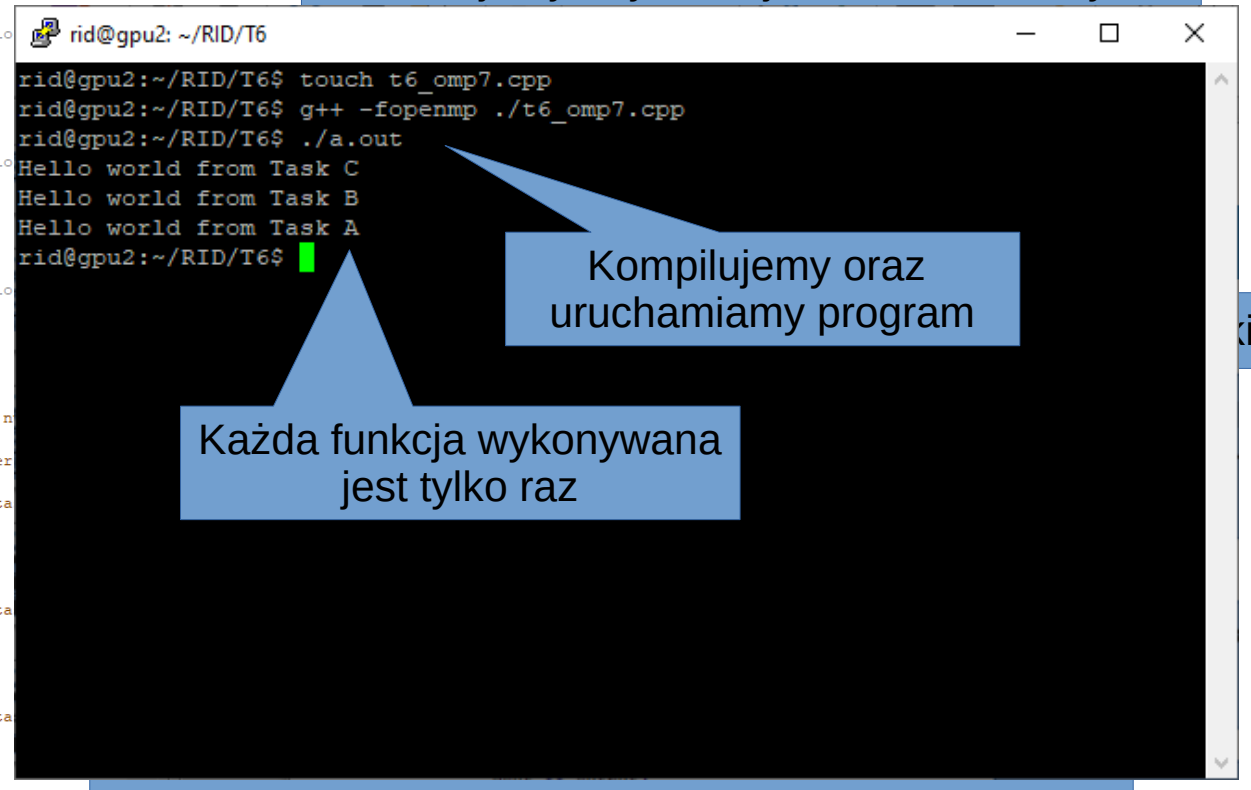
Każdy wątek pobiera zadanie z kolejki zadań wątku głównego i je wykonuje

W rezultacie, każda funkcja wykonana jest tylko raz

Przykład II – Tworzenie zadań

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

void taskA()
{
    string output = "Hello
    cout << output;
}

void taskB()
{
    string output = "Hello
    cout << output;
}

void taskC()
{
    string output = "Hello
    cout << output;
}

int main()
{
    #pragma omp parallel n
    {
        #pragma omp master
        {
            #pragma omp ta
            {
                taskA();
            }

            #pragma omp ta
            {
                taskB();
            }

            #pragma omp ta
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których



W rezultacie, każda funkcja wykonana jest tylko raz

Przykład III – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(0)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Przykład III – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(0)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla informacje o ID wątku, który ją realizuje

Przykład III – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(0)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla informacje o ID wątku, który ją realizuje

Tworzymy obszar równoległy zawierający trzy wątki

Przykład III – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(0)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla informacje o ID wątku, który ją realizuje

Tworzymy obszar równoległy zawierający trzy wątki

Każdy z wątków tworzy zadanie, które realizuje funkcję `taskA` oraz tworzy dwa zadania potomne wykonujące funkcje `taskB` i `taskC`

Przykład III – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(0)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla informacje o ID wątku, który ją realizuje

Tworzymy obszar równoległy zawierający trzy wątki

Zadanie nadrzędne tworzone jest z wykorzystaniem klauzuli `final` z wyrażeniem, które jest nieprawdziwe

Każdy z wątków tworzy zadanie, które realizuje funkcję `taskA` oraz tworzy dwa zadania potomne wykonujące funkcje `taskB` i `taskC`

Przykład III – Klauzula `final`

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

void taskA()
{
    string output = "T
    cout << output;
}

void taskB()
{
    string output = "T
    cout << output;
}

void taskC()
{
    string output = "T
    cout << output;
}

int main()
{
    #pragma omp parall
    {
        #pragma omp ta
        {
            taskA();

            #pragma om
            {
                taskB(
            }

            #pragma om
            {
                taskC(
            }
        }
    }

    return 0;
}
```

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp8.cpp
rid@gpu2:~/RID/T6$ ./a.out
Task A - watek #1
Task A - watek #2
Task B - watek #1
Task A - watek #0
Task B - watek #1
Task B - watek #1
Task C - watek #1
Task C - watek #2
Task C - watek #0
rid@gpu2:~/RID/T6$ ./a.out
Task A - watek #0
Task A - watek #1
Task B - watek #0
Task A - watek #2
Task B - watek #0
Task B - watek #0
Task C - watek #0
Task C - watek #1
Task C - watek #2
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Funkcja `taskA` wykonywana jest przez wątek tworzący zadanie

Zadania potomne realizowane są przez różne wątki

ch, których
adnia

formacje
izuje

ki

C

Przykład IV – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(1)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Przykład IV – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(1)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla informacje o ID wątku, który ją realizuje

Przykład IV – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(1)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla informacje o ID wątku, który ją realizuje

Tworzymy obszar równoległy zawierający trzy wątki

Przykład IV – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(1)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla informacje o ID wątku, który ją realizuje

Tworzymy obszar równoległy zawierający trzy wątki

Każdy z wątków tworzy zadanie, które realizuje funkcję `taskA` oraz tworzy dwa zadania potomne wykonujące funkcje `taskB` i `taskC`

Przykład IV – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskC()
{
    string output = "Task C - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp task final(1)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

Każda z funkcji wyświetla informacje o ID wątku, który ją realizuje

Tworzymy obszar równoległy zawierający trzy wątki

Zadanie nadrzędne tworzone jest z wykorzystaniem klauzuli `final` z wyrażeniem, które jest **prawdziwe**

Każdy z wątków tworzy zadanie, które realizuje funkcję `taskA` oraz tworzy dwa zadania potomne wykonujące funkcje `taskB` i `taskC`

Przykład IV – Klauzula `final`

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

void taskA()
{
    string output = "Task A - watek #" + to_string(omp_get_thread_num()) + "\n";
    cout << output;
}

void taskB()
{
    string output = "Task B - watek
    cout << output;
}

void taskC()
{
    string output = "Task C - watek
    cout << output;
}

int main()
{
    #pragma omp parallel num_threads
    {
        #pragma omp task final(1)
        {
            taskA();

            #pragma omp task
            {
                taskB();
            }

            #pragma omp task
            {
                taskC();
            }
        }
    }

    return 0;
}
```

Definiujemy trzy funkcje w ramach, których realizowane są „różne” zadnia

rid@gpu2: ~/RID/T6

```
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp9.cpp
rid@gpu2:~/RID/T6$ ./a.out
Task A - watek #0
Task A - watek #2
Task A - watek #1
Task B - watek #0
Task C - watek #0
Task B - watek #1
Task B - watek #2
Task C - watek #2
Task C - watek #1
rid@gpu2:~/RID/T6$ ./a.out
Task A - watek #0
Task A - watek #2
Task A - watek #1
Task B - watek #0
Task C - watek #0
Task B - watek #1
Task C - watek #1
Task B - watek #2
Task C - watek #2
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Zastosowanie klauzuli `final(1)` powoduje, że zadanie nadrzędne oraz zadania potomne wykonywane są przez ten sam wątek

rmacje
je

Przykład V – Klauzula depend

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

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, result3;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task depend(out:result1)
                result1 = fun1();

            #pragma omp task depend(out:result2)
                result2 = fun2();

            #pragma omp task depend(in:result1, result2) depend(out:result3)
                result3 = 2 * (result1 + result2);

            #pragma omp task depend(in:result3)
            {
                string output = "Wynik: " + to_string(result3) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Przykład V – Klauzula depend

Funkcje fun1() oraz fun2() wykonują ”czasochłonne obliczenia”

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

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, result3;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task depend(out:result1)
                result1 = fun1();

            #pragma omp task depend(out:result2)
                result2 = fun2();

            #pragma omp task depend(in:result1, result2) depend(out:result3)
                result3 = 2 * (result1 + result2);

            #pragma omp task depend(in:result3)
            {
                string output = "Wynik: " + to_string(result3) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Przykład V – Klauzula depend

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

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, result3;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task depend(out:result1)
                result1 = fun1();

            #pragma omp task depend(out:result2)
                result2 = fun2();

            #pragma omp task depend(in:result1, result2) depend(out:result3)
                result3 = 2 * (result1 + result2);

            #pragma omp task depend(in:result3)
            {
                string output = "Wynik: " + to_string(result3) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Funkcje fun1() oraz fun2() wykonują "czasochłonne obliczenia"

Wyniki obliczeń zwracane są po upływie 1 sekundy

Przykład V – Klauzula depend

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

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, result3;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task depend(out:result1)
                result1 = fun1();

            #pragma omp task depend(out:result2)
                result2 = fun2();

            #pragma omp task depend(in:result1, result2) depend(out:result3)
                result3 = 2 * (result1 + result2);

            #pragma omp task depend(in:result3)
            {
                string output = "Wynik: " + to_string(result3) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Funkcje fun1() oraz fun2() wykonują "czasochłonne obliczenia"

Wyniki obliczeń zwracane są po upływie 1 sekundy

Tworzymy obszar równoległy zawierający trzy wątki

Przykład V – Klauzula depend

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

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, result3;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task depend(out:result1)
                result1 = fun1();

            #pragma omp task depend(out:result2)
                result2 = fun2();

            #pragma omp task depend(in:result1, result2) depend(out:result3)
                result3 = 2 * (result1 + result2);

            #pragma omp task depend(in:result3)
            {
                string output = "Wynik: " + to_string(result3) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Funkcje fun1() oraz fun2() wykonują "czasochłonne obliczenia"

Wyniki obliczeń zwracane są po upływie 1 sekundy

Tworzymy obszar równoległy zawierający trzy wątki

Wątek główny tworzy zadania, które są od siebie zależne

Przykład V – Klauzula depend

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

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, result3;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task depend(out:result1)
                result1 = fun1();

            #pragma omp task depend(out:result2)
                result2 = fun2();

            #pragma omp task depend(in:result1, result2) depend(out:result3)
                result3 = 2 * (result1 + result2);

            #pragma omp task depend(in:result3)
            {
                string output = "Wynik: " + to_string(result3) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Funkcje fun1() oraz fun2() wykonują "czasochłonne obliczenia"

Wyniki obliczeń zwracane są po upływie 1 sekundy

Tworzymy obszar równoległy zawierający trzy wątki

Wątek główny tworzy zadania, które są od siebie zależne

Pierwsze dwa zadania są wykonywane równolegle

Przykład V – Klauzula depend

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

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, result3;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task depend(out:result1)
                result1 = fun1();

            #pragma omp task depend(out:result2)
                result2 = fun2();

            #pragma omp task depend(in:result1, result2) depend(out:result3)
                result3 = 2 * (result1 + result2);

            #pragma omp task depend(in:result3)
            {
                string output = "Wynik: " + to_string(result3) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Funkcje `fun1()` oraz `fun2()` wykonują "czasochłonne obliczenia"

Wyniki obliczeń zwracane są po upływie 1 sekundy

Tworzymy obszar równoległy zawierający trzy wątki

Wątek główny tworzy zadania, które są od siebie zależne

Pierwsze dwa zadania są wykonywane równolegle

Trzecie zadanie zależy od wyniku wykonania poprzednich dwóch zadań

Przykład V – Klauzula depend

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

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, result3;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp task depend(out:result1)
                result1 = fun1();

            #pragma omp task depend(out:result2)
                result2 = fun2();

            #pragma omp task depend(in:result1, result2) depend(out:result3)
                result3 = 2 * (result1 + result2);

            #pragma omp task depend(in:result3)
            {
                string output = "Wynik: " + to_string(result3) + "\n";
                cout << output;
            }
        }
    }

    return 0;
}
```

Funkcje fun1() oraz fun2() wykonują "czasochłonne obliczenia"

Wyniki obliczeń zwracane są po upływie 1 sekundy

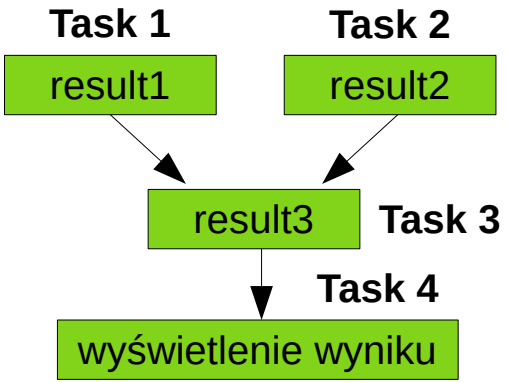
Tworzymy obszar równoległy zawierający trzy wątki

Wątek główny tworzy zadania, które są od siebie zależne

Pierwsze dwa zadania są wykonywane równolegle

Trzecie zadanie zależy od wyniku wykonania poprzednich dwóch zadań

Ostatnie zadanie zależy od wyniku trzeciego zadania



Przykład V – Klauzula depend

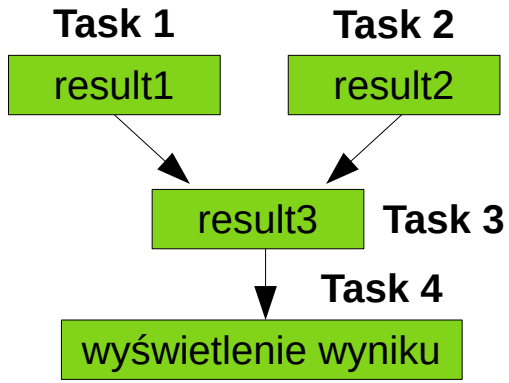
```
#include <iostream>
#include <omp.h>
#include <string>
#include <unistd.h>
using namespace std;

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, r
    #pragma omp parallel nu
    {
        #pragma omp master
        {
            #pragma omp tas
                result1 = f

            #pragma omp tas
                result2 = f

            #pragma omp tas
                result3 = 2

            #pragma omp tas
            {
                string outp
                cout << out
            }
        }
    }

    return 0;
}
```

Funkcje fun1() oraz fun2() wykonują "czasochłonne obliczenia"

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp10.cpp
rid@gpu2:~/RID/T6$ ./a.out
Wynik: 36
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Otrzymujemy wynik zgodny z oczekiwaniem



re są od

e równolegle

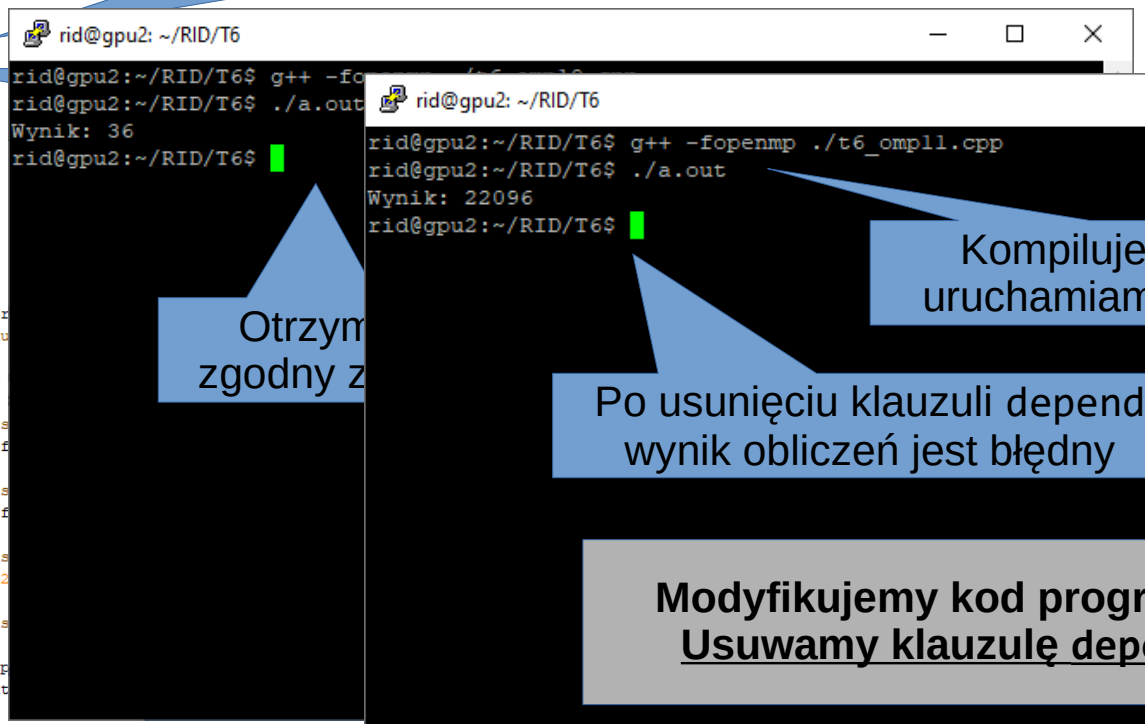
l wyniku wykonania
ch zadań

Ostatnie zadanie zależy od wyniku trzeciego zadania

Przykład V – Klauzula depend

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

int fun1()
{
    sleep(1);
    int value = 3;
    return value;
}

int fun2()
{
    sleep(1);
    int value = 15;
    return value;
}

int main()
{
    int result1, result2, r
    #pragma omp parallel nu
    {
        #pragma omp master
        {
            #pragma omp tas
                result1 = f

            #pragma omp tas
                result2 = f

            #pragma omp tas
                result3 = 2

            #pragma omp tas
            {
                string outp
                cout << out
            }
        }
    }

    return 0;
}
```

Funkcje fun1() oraz fun2() wykonują "czasochłonne obliczenia"

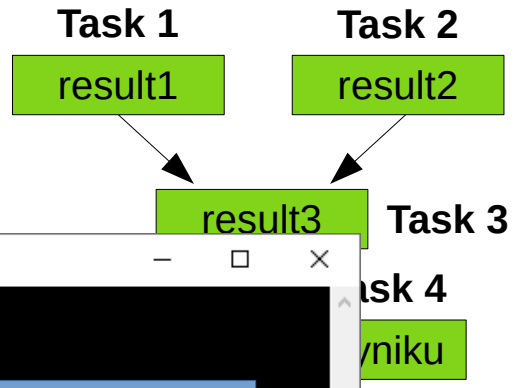


Otrzym
zgodny z

Kompilujemy oraz uruchamiamy program

Po usunięciu klauzuli depend wynik obliczeń jest błędny

Modyfikujemy kod programu
<u>Usuwamy klauzulę depend</u>



Przykład VI – Klauzula depend

Algorytm blokowego mnożenia macierzy

```
// Założenie: N dzieli się bez reszty przez BS
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
    for (int i = 0; i < N; i+=BS)
    {
        for (int j = 0; j < N; j+=BS)
        {
            for (int k = 0; k < N; k+=BS)
            {
                // Uwaga 1: i, j, k, A, B, C są domyślnie firsprivate
                // Uwaga 2: A, B oraz C są wskaźnikami
                #pragma omp task depend( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
                    depend( inout: C[i:BS][j:BS] )
                    for (int ii = i; ii < i+BS; ii++ )
                        for (int jj = j; jj < j+BS; jj++ )
                            for (int kk = k; kk < k+BS; kk++ )
                                C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
            }
        }
    }
}
```

Przykład VI – Klauzula depend

Algorytm blokowego mnożenia macierzy

```
// Założenie: N dzieli się bez reszty przez BS
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
    for (int i = 0; i < N; i+=BS)
    {
        for (int j = 0; j < N; j+=BS)
        {
            for (int k = 0; k < N; k+=BS)
            {
                // Uwaga 1: i, j, k, A, B, C są domyślnie firsprivate
                // Uwaga 2: A, B oraz C są wskaźnikami
                #pragma omp task depend( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
                    depend( inout: C[i:BS][j:BS] )
                    for (int ii = i; ii < i+BS; ii++ )
                        for (int jj = j; jj < j+BS; jj++ )
                            for (int kk = k; kk < k+BS; kk++ )
                                C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
            }
        }
    }
}
```

Klauzula depend oprócz zwykłych zmiennych może zawierać także sekcje tablic lub całe tablice

Przykład VI – Klauzula depend

Algorytm blokowego mnożenia macierzy

```
// Założenie: N dzieli się bez reszty przez BS
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
    for (int i = 0; i < N; i+=BS)
    {
        for (int j = 0; j < N; j+=BS)
        {
            for (int k = 0; k < N; k+=BS)
            {
                // Uwaga 1: i, j, k, A, B, C są domyślnie firsprivate
                // Uwaga 2: A, B oraz C są wskaźnikami
                #pragma omp task depend( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
                    depend( inout: C[i:BS][j:BS] )
                    for (int ii = i; ii < i+BS; ii++ )
                        for (int jj = j; jj < j+BS; jj++ )
                            for (int kk = k; kk < k+BS; kk++ )
                                C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
            }
        }
    }
}
```

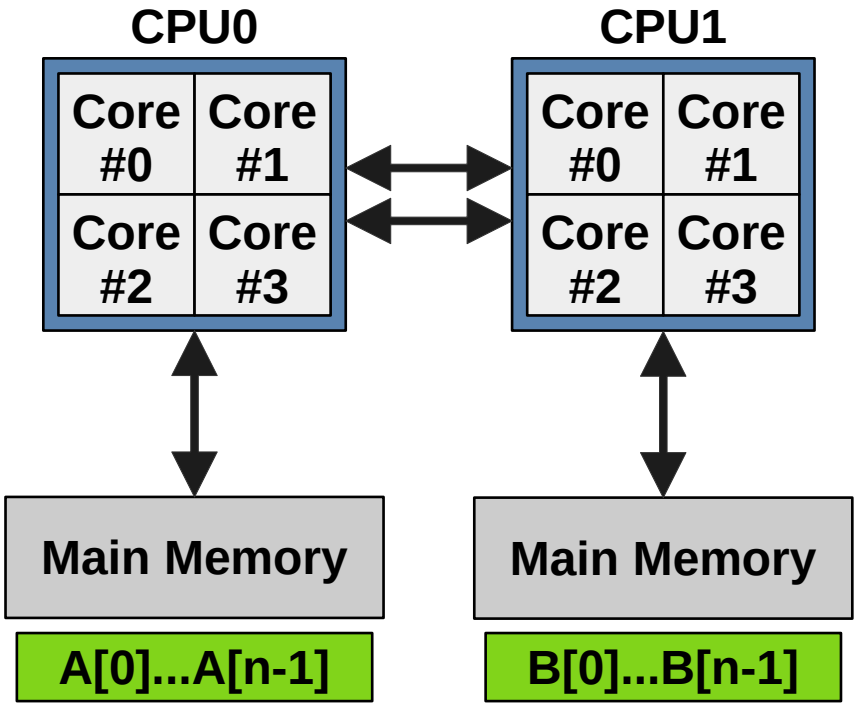
Macierze dzielone są na mniejsze bloki w ramach, których wykonywane są obliczenia

Klauzula depend oprócz zwykłych zmiennych może zawierać także sekcje tablic lub całe tablice

Klauzula depend zastosowana została, aby zapewnić poprawną realizację obliczeń w ramach poszczególnych bloków macierzy C

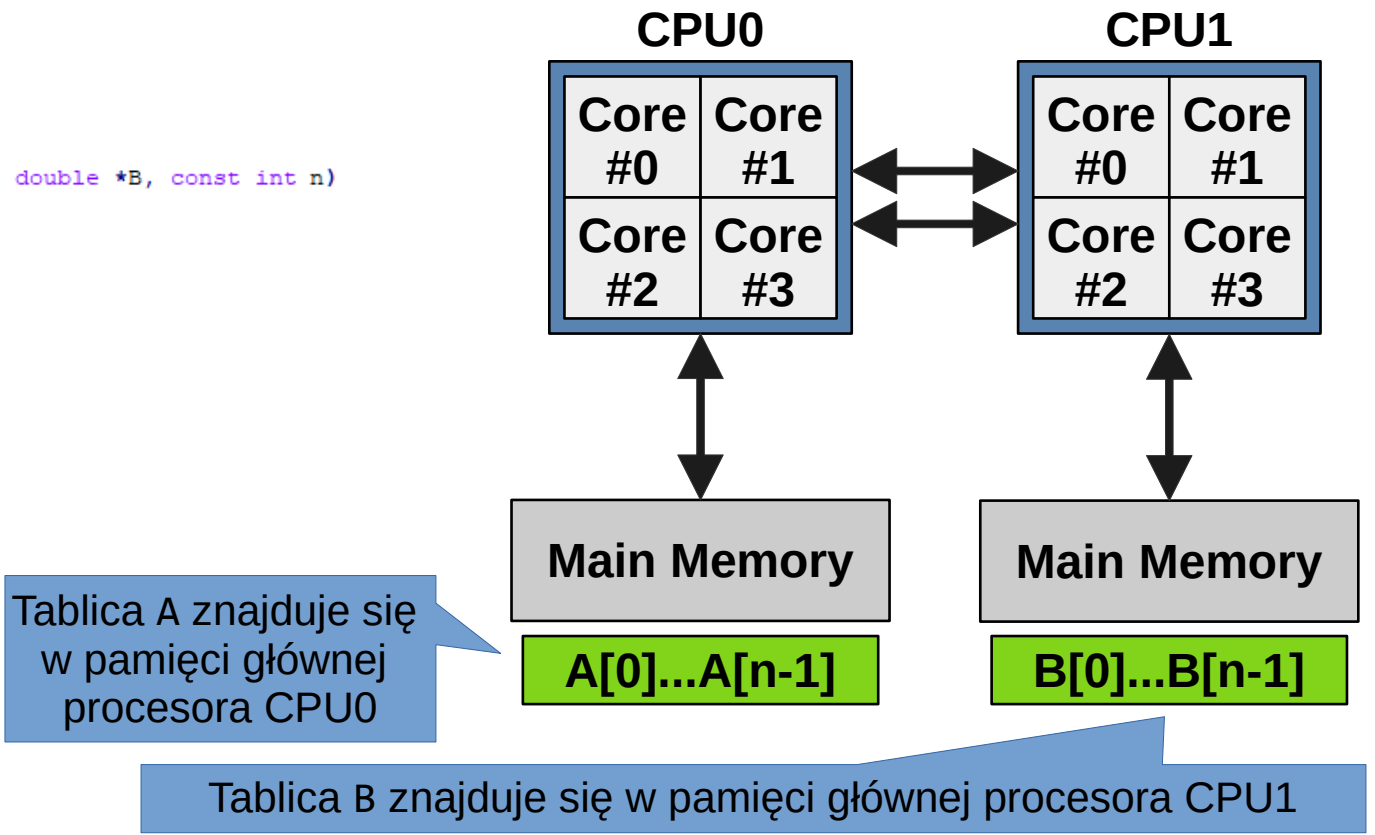
Przykład VII – Klauzula affinity

```
void task_affinity_example(const double *A, const double *B, const int n)
{
    #pragma omp task affinity(A[0:n])
    {
        do_important_computations(A);
    }
    #pragma omp task affinity(B[0:n])
    {
        do_important_computations(B);
    }
}
```



Przykład VII – Klauzula `affinity`

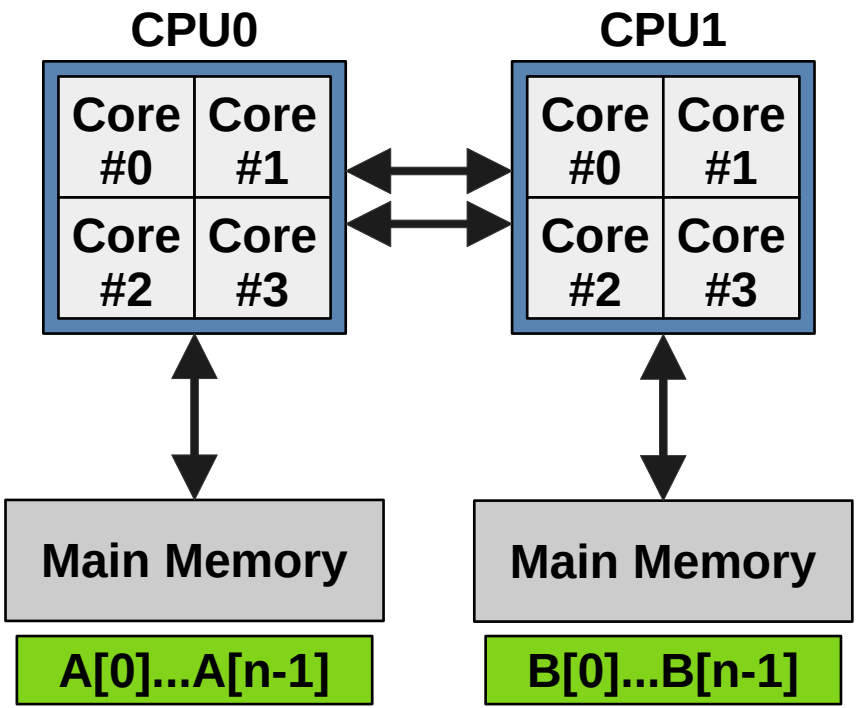
```
void task_affinity_example(const double *A, const double *B, const int n)
{
    #pragma omp task affinity(A[0:n])
    {
        do_important_computations(A);
    }
    #pragma omp task affinity(B[0:n])
    {
        do_important_computations(B);
    }
}
```



Przykład VII – Klauzula `affinity`

```
void task_affinity_example(const double *A, const double *B, const int n)
{
    #pragma omp task affinity(A[0:n])
    {
        do_important_computations(A);
    }
    #pragma omp task affinity(B[0:n])
    {
        do_important_computations(B);
    }
}
```

Tworzymy dwa zadania wykonujące obliczenia na tablicach A oraz B



Tablica A znajduje się w pamięci głównej procesora CPU0

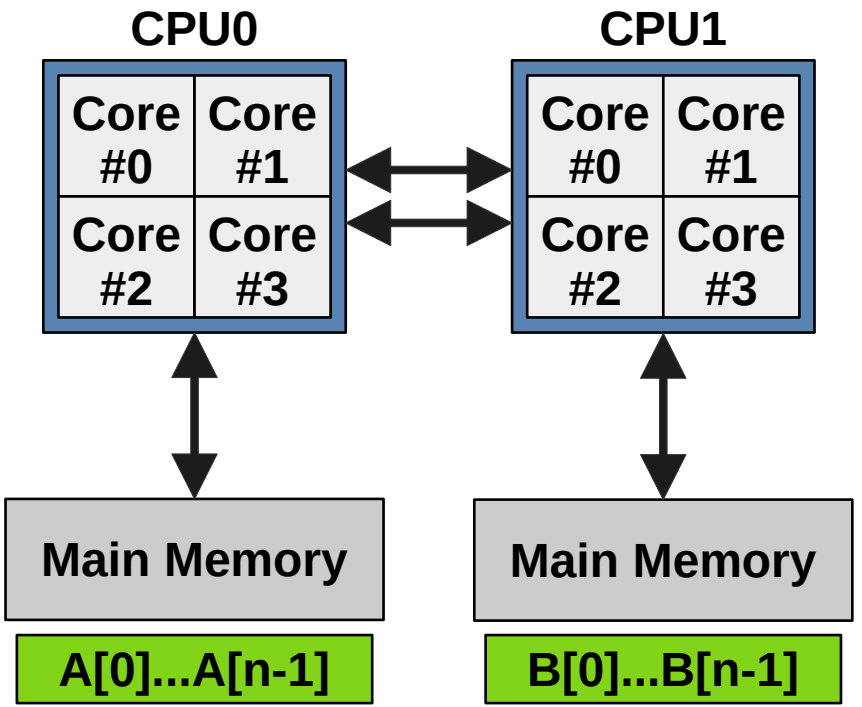
Tablica B znajduje się w pamięci głównej procesora CPU1

Przykład VII – Klauzula `affinity`

```
void task_affinity_example(const double *A, const double *B, const int n)
{
    #pragma omp task affinity(A[0:n])
    {
        do_important_computations(A);
    }
    #pragma omp task affinity(B[0:n])
    {
        do_important_computations(B);
    }
}
```

Tworzymy dwa zadania wykonujące obliczenia na tablicach `A` oraz `B`

Chcemy aby zadania wykonane zostały przez wątki znajdujące się jak najbliżej danych. W tym celu stosujemy klauzule `affinity`.



Tablica `A` znajduje się w pamięci głównej procesora CPU0

Tablica `B` znajduje się w pamięci głównej procesora CPU1

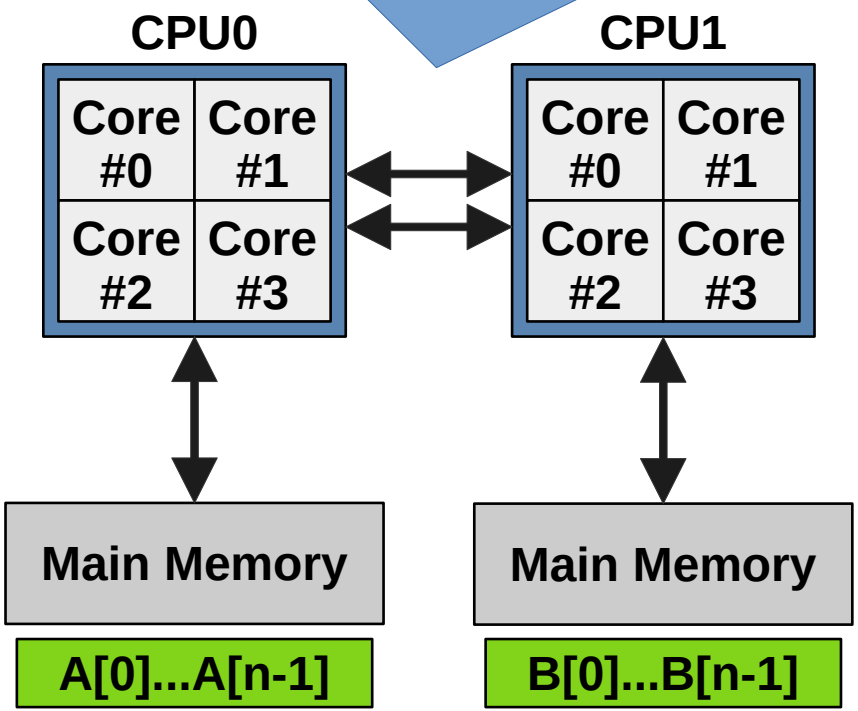
Przykład VII – Klauzula `affinity`

Zadanie wykorzystujące tablicę **A** zostanie wykonane przez wątek procesora **CPU0**, natomiast zadanie przetwarzające tablicę B przez wątek procesora **CPU1**

```
void task_affinity_example(const double *A, const double *B, const int n)
{
    #pragma omp task affinity(A[0:n])
    {
        do_important_computations(A);
    }
    #pragma omp task affinity(B[0:n])
    {
        do_important_computations(B);
    }
}
```

Tworzymy dwa zadania wykonujące obliczenia na tablicach A oraz B

Chcemy aby zadania wykonane zostały przez wątki znajdujące się jak najbliżej danych. W tym celu stosujemy klauzule `affinity`.



Tablica A znajduje się w pamięci głównej procesora CPU0

Tablica B znajduje się w pamięci głównej procesora CPU1

Przykład VII – Klauzula `affinity`

Zadanie wykorzystujące tablicę **A** zostanie wykonane przez wątek procesora **CPU0**, natomiast zadanie przetwarzające tablicę B przez wątek procesora **CPU1**

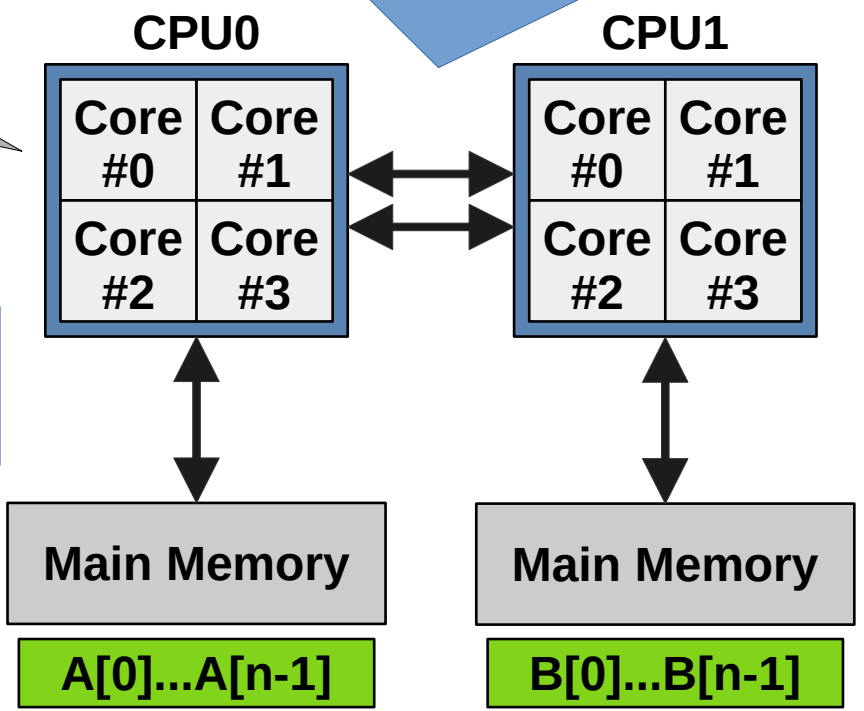
Wciąż jednak istnieje ryzyko kradzieży zadania realizującego obliczenia na tablicy A przez wątki procesora CPU1 w przypadku, kiedy wszystkie wątki CPU0 są zajęte (i odwrotnie)

```
void task_affinity_example(const double *A, const double *B, const int n)
{
    #pragma omp task affinity(A[0:n])
    {
        do_important_computations(A);
    }
    #pragma omp task affinity(B[0:n])
    {
        do_important_computations(B);
    }
}
```

Tworzymy dwa zadania wykonujące obliczenia na tablicach A oraz B

Chcemy aby zadania wykonane zostały przez wątki znajdujące się jak najbliżej danych. W tym celu stosujemy klauzule `affinity`.

Tablica A znajduje się w pamięci głównej procesora CPU0



Tablica B znajduje się w pamięci głównej procesora CPU1

Przykład VIII – Zrównoleglenie pętli

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Przykład VIII – Zrównoleglenie pętli

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Przykład VIII – Zrównoleglenie pętli

Tworzymy dwie listy z dowiązaniami

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Przykład VIII – Zrównoleglenie pętli

Tworzymy dwie listy z dowiązaniami

Obie listy wypełniamy w ten sam sposób

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Przykład VIII – Zrównoleglenie pętli

Tworzymy dwie listy z dowiązaniami

Obie listy wypełniamy w ten sam sposób

Kod sekwencyjny – lista przetwarzana jest przez jeden wątek

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Przykład VIII – Zrównoleglenie pętli

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Tworzymy dwie listy z dowiązaniami

Obie listy wypełniamy w ten sam sposób

Kod sekwencyjny – lista przetwarzana jest przez jeden wątek

Po elementach listy poruszamy się wykorzystując pętlę `while`

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

Przykład VIII – Zrównoleglenie pętli

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Tworzymy dwie listy z dowiązaniami

Obie listy wypełniamy w ten sam sposób

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

Kod sekwencyjny – lista przetwarzana jest przez jeden wątek

Po elementach listy poruszamy się wykorzystując pętlę `while`

Zrównoleglamy proces przetwarzania elementów

Przykład VIII – Zrównoleglenie pętli

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Tworzymy dwie listy z dowiązaniami

Obie listy wypełniamy w ten sam sposób

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

Kod sekwencyjny – lista przetwarzana jest przez jeden wątek

Po elementach listy poruszamy się wykorzystując pętlę `while`

Zrównoleglamy proces przetwarzania elementów

Tworzymy cztery wątki

Przykład VIII – Zrównoleglenie pętli

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Tworzymy dwie listy z dowiązaniami

Obie listy wypełniamy w ten sam sposób

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

Kod sekwencyjny – lista przetwarzana jest przez jeden wątek

Po elementach listy poruszamy się wykorzystując pętlę `while`

Zrównoleglamy proces przetwarzania elementów

Tworzymy cztery wątki

OpenMP nie dostarcza mechanizmów umożliwiających zrównoleglenie pętli `while`

Przykład VIII – Zrównoleglenie pętli

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Tworzymy dwie listy z dowiązaniami

Obie listy wypełniamy w ten sam sposób

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

Kod sekwencyjny – lista przetwarzana jest przez jeden wątek

Po elementach listy poruszamy się wykorzystując pętlę `while`

Zrównoleglamy proces przetwarzania elementów

Tworzymy cztery wątki

OpenMP nie dostarcza mechanizmów umożliwiających zrównoleglenie pętli `while`

Jednym ze sposób zrównoleglenia jest pętli `while` jest wykorzystanie zadań OpenMP

Przykład VIII – Zrównoleglenie pętli

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Tworzymy dwie listy z dowiązaniami

Obie listy wypełniamy w ten sam sposób

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

Kod sekwencyjny – lista przetwarzana jest przez jeden wątek

Po elementach listy poruszamy się wykorzystując pętlę `while`

Zrównoleglamy proces przetwarzania elementów

Tworzymy cztery wątki

OpenMP nie dostarcza mechanizmów umożliwiających zrównoleglenie pętli `while`

Jednym ze sposób zrównoleglenia jest pętli `while` jest wykorzystanie zadań OpenMP

Wątek główny tworzy zadania. Każde zadanie przetwarza pojedynczy element listy

Przykład VIII – Zrównoleglenie pętli

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_list);

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyjny..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    while(curr)
    {
        process(curr);
        curr = curr->next;
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n\n";

    cout << "Kod rownolegly..." << endl;
    start = omp_get_wtime();
    curr = serial_list->begin();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            while(curr)
            {
                #pragma omp task firstprivate(curr)
                    process(curr);
                curr = curr->next;
            }
        }
    }
    stop = omp_get_wtime();
    cout << "Czas przetwarzania: " << stop-start << " [s]\n";

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

Tworzymy dwie listy z dowiązaniami

Obie listy wypełniamy w ten sam sposób

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

Kod sekwencyjny – lista przetwarzana jest przez jeden wątek

Po elementach listy poruszamy się wykorzystując pętlę `while`

Zrównoleglamy proces przetwarzania elementów

Tworzymy cztery wątki

OpenMP nie dostarcza mechanizmów umożliwiających zrównoleglenie pętli `while`

Jednym ze sposób zrównoleglenia jest pętli `while` jest wykorzystanie zadań OpenMP

Wątek główny tworzy zadania. Każde zadanie przetwarza pojedynczy element listy

Dla obu wersji mierzymy czas wykonania

Przykład VIII – Zrównoleglenie pętli

Tworzymy dwie listy z dowiązaniami

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_li

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyj
    start = omp_get_wtime
    curr = serial_list->b
    while(curr)
    {
        process(curr);
        curr = curr->next
    }
    stop = omp_get_wtime()
    cout << "Czas przetwa

    cout << "Kod rownoleg
    start = omp_get_wtime
    curr = serial_list->b
    #pragma omp parallel
    {
        #pragma omp maste
        {
            while(curr)
            {
                #pragma o
                    proce
                curr = cu
            }
        }
    }
    stop = omp_get_wtime()
    cout << "Czas przetwa

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp12.cpp
rid@gpu2:~/RID/T6$ ./a.out
Kod sekwencyjny...
English: Hello World! - ENGLISH: HELLO WORLD!
French: Bonjour, le monde! - FRENCH: BONJOUR, LE MONDE!
Spanish: Hola al mundo - SPANISH: HOLA AL MUNDO
Klingon: Nuq neH! - KLINGON: NUQ NEH!
German: Guten Tag, Welt! - GERMAN: GUTEN TAG, WELT!
Russian: Zdravstvytye, mir! - RUSSIAN: ZDRAVSTVYTYE, MIR!
Japan: Sekai e konnichiwa! - JAPAN: SEKAI E KONNICHIWA!
Latin: Orbis, te saluto! - LATIN: ORBIS, TE SALUTO!
Czas przetwarzania: 8.00118 [s]

Kod rownolegly...
Klingon: Nuq neH! - KLINGON: NUQ NEH!
English: Hello World! - ENGLISH: HELLO WORLD!
French: Bonjour, le monde! - FRENCH: BONJOUR, LE MONDE!
Spanish: Hola al mundo - SPANISH: HOLA AL MUNDO
German: Guten Tag, Welt! - GERMAN: GUTEN TAG, WELT!
Russian: Zdravstvytye, mir! - RUSSIAN: ZDRAVSTVYTYE, MIR!
Japan: Sekai e konnichiwa! - JAPAN: SEKAI E KONNICHIWA!
Latin: Orbis, te saluto! - LATIN: ORBIS, TE SALUTO!
Czas przetwarzania: 2.00101 [s]
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Obie listy wypełnione są tak samo

Wątki modyfikują napisy, tak aby wszystkie zostały napisane dużymi literami a następnie są usypiane na 1 sekundę

Równoległe przetwarzanie pętli za pomocą zadań finalnie skraca czas przetwarzania z 8 sekund do 2 sekund

przetwarza pojedynczy element listy

czas wykonania

Przykład VIII – Zrównoleglenie pętli

Tworzymy dwie listy z dowiązaniami

Ze względu na rozmiar pliku z kodem źródłowym zaprezentowana została jedynie funkcja **`main()`**. Cały program znajduje się w pliku ***`t6_omp12.cpp`***.

```
int main()
{
    List *serial_list = new List();
    List *parallel_list = new List();
    fill_list(serial_list);
    fill_list(parallel_li

    double start, stop;
    Node *curr;

    cout << "Kod sekwencyj
    start = omp_get_wtime
    curr = serial_list->b
    while(curr)
    {
        process(curr);
        curr = curr->next
    }
    stop = omp_get_wtime()
    cout << "Czas przetwa

    cout << "Kod rownoleg
    start = omp_get_wtime
    curr = serial_list->b
    #pragma omp parallel
    {
        #pragma omp maste
        {
            while(curr)
            {
                #pragma o
                    proce
                curr = cu
            }
        }
    }
    stop = omp_get_wtime()
    cout << "Czas przetwa

    delete parallel_list;
    delete serial_list;
    return 0;
}
```

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp12.cpp
rid@gpu2:~/RID/T6$ ./a.out
Kod sekwencyjny...
English: Hello World! - ENGLISH: HELLO WORLD!
French: Bonjour, le monde! - FRENCH: BONJOUR, LE MONDE!
Spanish: Hola al mundo - SPANISH: HOLA AL MUNDO
Klingon: Nuq neH! - KLINGON: NUQ NEH!
German: Guten Tag, Welt! - GERMAN: GUTEN TAG, WELT!
Russian: Zdravstvytye, mir! - RUSSIAN: ZDRAVSTVYTYE, MIR!
Japan: Sekai e konnichiwa! - JAPAN: SEKAI E KONNICHIWA!
Latin: Orbis, te saluto! - LATIN: ORBIS, TE SALUTO!
Czas przetwarzania

Kod rownolegly...
Klingon: Nuq neH!
English: Hello Wor
French: Bonjour, l
Spanish: Hola al m
German: Guten Tag,
Russian: Zdravstvy
Japan: Sekai e kon
Latin: Orbis, te saluto! - LATIN: ORBIS, TE SALUTO!
Czas przetwarzania: 2.00101 [s]
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Obie listy wypełnione są tak samo

Zadania OpenMP umożliwiają równoległe wykonanie pętli, które nie mają odpowiedniej struktury iteracji, aby zrównoleglić je z wykorzystaniem dyrektywy `#pragma omp for`

Równoległe przetwarzanie pętli za pomocą zadań finalnie skraca czas przetwarzania z 8 sekund do 2 sekund

przetwarza pojedynczy element listy

czas wykonania

Synchronizacja zdań

- Synchronizacja zadań w standardzie OpenMP możliwa jest poprzez wykorzystanie następujących konstrukcji:
  - Konstrukcji `taskwait`
  - Dyrektywy `#pragma omp barrier`
  - Konstrukcji `taskgroup`

Konstrukcje `taskwait` oraz `barrier`

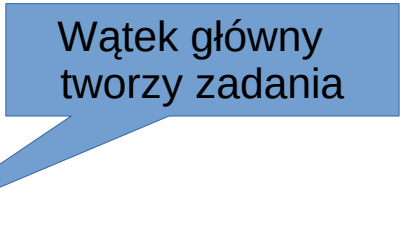
- Konstrukcja `taskwait` wstrzymuje wykonanie zadania, które ją wywołuje do czasu zakończenia realizacji utworzonych przez nie zadań
  - Może zostać zastosowana jedynie do zadań wygenerowanych przez bieżące zadanie, nie działa dla zadań „potomnych”
  - Kod wykonywany przez wątek w regionie równoległym jest tutaj uważany za zadanie
- Dyrektywa `#pragma omp barrier` (lub barriera niejawna) wymusza zakończenie wykonywania wszystkich zadań utworzonych przed jej wystąpieniem

Przykład IX – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                sleep(1);
                cout << "Task B\n";
            }

            #pragma omp task
            {
                sleep(5);
                cout << "Task C\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Przykład IX – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                sleep(1);
                cout << "Task B\n";
            }

            #pragma omp task
            {
                sleep(5);
                cout << "Task C\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```



Przykład IX – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                sleep(1);
                cout << "Task B\n";
            }

            #pragma omp task
            {
                sleep(5);
                cout << "Task C\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

Zdania usypiane są na różny okres czasu

Przykład IX – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                sleep(1);
                cout << "Task B\n";
            }

            #pragma omp task
            {
                sleep(5);
                cout << "Task C\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

Zdania usypiane są na różny okres czasu

Konstrukcja `taskwait` wywołana jest po utworzeniu zada `taskA, taskB` oraz `taskC`

Przykład IX – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                sleep(1);
                cout << "Task B\n";
            }

            #pragma omp task
            {
                sleep(5);
                cout << "Task C\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

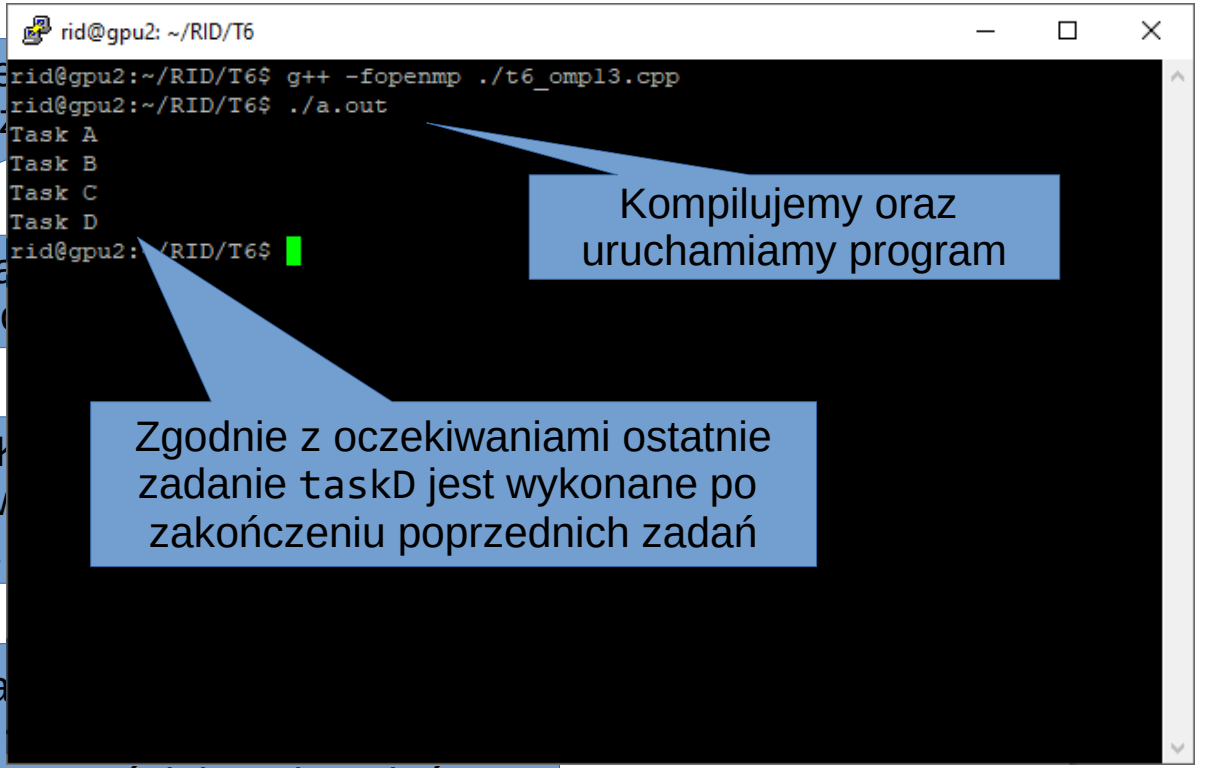
Zdania usypiane są na różny okres czasu

Konstrukcja `taskwait` wywołana jest po utworzeniu zada `taskA, taskB` oraz `taskC`

Zadanie `taskD` utworzone jest po zakończeniu realizacji wcześniejszych zadań

Przykład IX – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                sleep(1);
                cout << "Task B\n";
            }

            #pragma omp task
            {
                sleep(5);
                cout << "Task C\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```



Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

Zadania usypiane są na różny okres czasu

Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

Zadania usypiane są na różny okres czasu

Zadanie `taskB` tworzy zadanie `taskC`

Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

Zadania usypiane są na różny okres czasu

Zadanie `taskB` tworzy zadanie `taskC`

Zadanie `taskC` uśpione jest najdłużej ze wszystkich zadań

Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

Zadania usypiane są na różny okres czasu

Zadanie `taskB` tworzy zadanie `taskC`

Zadanie `taskC` uśpione jest najdłużej ze wszystkich zadań

Konstrukcja taskwait zadziała tylko dla zadań `taskA` oraz `taskB`

Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

Zadania usypiane są na różny okres czasu

Zadanie `taskB` tworzy zadanie `taskC`

Zadanie `taskC` uśpione jest najdłużej ze wszystkich zadań

Konstrukcja taskwait zadziała tylko dla zadań `taskA` oraz `taskB`

Zadanie `taskD` wykonane zostanie po zakończaniu `taskA` oraz `taskB`

Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

Zadania usypiane są na różny okres czasu

Zadanie `taskB` tworzy zadanie `taskC`

Zadanie `taskC` uśpione jest najdłużej ze wszystkich zadań

Konstrukcja taskwait zadziała tylko dla zadań `taskA` oraz `taskB`

W tym samym czasie, zadanie `taskC` wciąż jest uśpione

Zadanie `taskD` wykonane zostanie po zakończaniu `taskA` oraz `taskB`

Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pragma omp task
            {
                cout << "Task D\n";
            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zadania

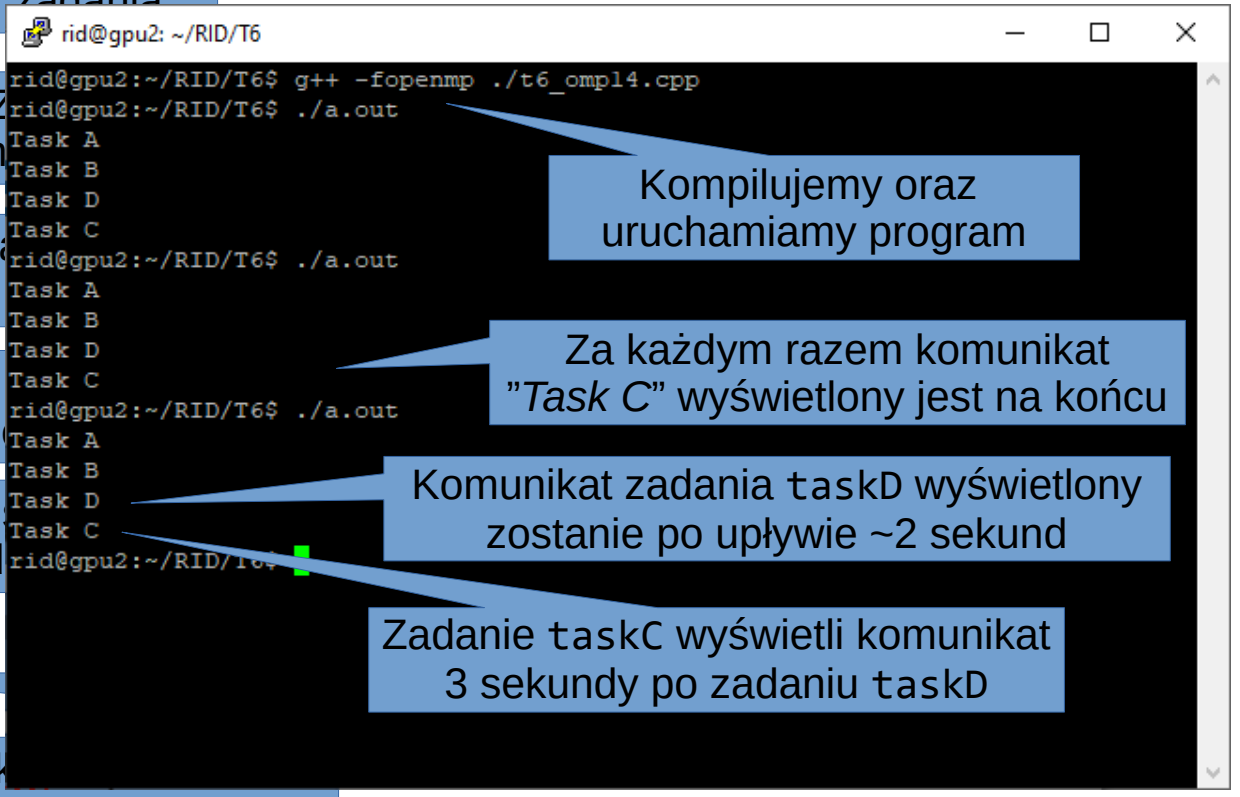
Z
są n

Z

j

Kon
d

Zadanie `taskD` wyk
po zakończaniu `taskA` oraz `taskB`



Przykład X – konstrukcja `taskwait`

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

int main()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                sleep(1);
                cout << "Task A\n";
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    sleep(5);
                    cout << "Task C\n";
                }

                sleep(2);
                cout << "Task B\n";
            }

            #pragma omp taskwait

            #pr
            {

            }
        }
    }

    return 0;
}
```

Wątek główny tworzy zada

Z są n

Z

j

Kon d

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++ -fopenmp ./t6_omp14.cpp
rid@gpu2:~/RID/T6$ ./a.out
Task A
Task B
Task D
Task C
rid@gpu2:~/RID/T6$ ./a.out
Task A
Task B
Task D
Task C
rid@gpu2:~/RID/T6$ ./a.out
Task A
Task B
Task D
Task C
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Za każdym razem komunikat "*Task C*" wyświetlony jest na końcu

Komunikat zadania `taskD` wyświetlony zostanie po upływie ~2 sekund

Zadanie `taskC` wyświetli komunikat 3 sekundy po zadaniu `taskD`

Konstrukcja `taskwait` nie działa na rzecz zadania potomnego `taskC`

Synchronizacja zadań – konstrukcja `barrier`

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            #pragma omp task
            {
                taskB();
            }

            taskA();
        }

        #pragma omp task
        {
            #pragma omp task
            {
                taskD();
            }

            taskC();
        }
    }

    #pragma omp barrier

    #pragma omp master
    {
        #pragma omp task
        {
            taskE();
        }
    }
}
```

Synchronizacja zadań – konstrukcja `barrier`

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            #pragma omp task
            {
                taskB();
            }

            taskA();
        }

        #pragma omp task
        {
            #pragma omp task
            {
                taskD();
            }

            taskC();
        }
    }

    #pragma omp barrier

    #pragma omp master
    {
        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zadania

Synchronizacja zadań – konstrukcja `barrier`

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            #pragma omp task
            {
                taskB();
            }

            taskA();
        }

        #pragma omp task
        {
            #pragma omp task
            {
                taskD();
            }

            taskC();
        }
    }

    #pragma omp barrier

    #pragma omp master
    {
        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zadania

Wątek główny tworzy dwa zadania

Synchronizacja zadań – konstrukcja `barrier`

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            #pragma omp task
            {
                taskB();
            }

            taskA();
        }

        #pragma omp task
        {
            #pragma omp task
            {
                taskD();
            }

            taskC();
        }
    }

    #pragma omp barrier

    #pragma omp master
    {
        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zadania

Wątek główny tworzy dwa zadania

Każde z zadań tworzy zdania potomne

Synchronizacja zadań – konstrukcja `barrier`

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            #pragma omp task
            {
                taskB();
            }

            taskA();
        }

        #pragma omp task
        {
            #pragma omp task
            {
                taskD();
            }

            taskC();
        }
    }

    #pragma omp barrier

    #pragma omp master
    {
        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zadania

Wątek główny tworzy dwa zadania

Każde z zadań tworzy zdania potomne

Wychodzimy z obszaru realizowanego przez wątek główny

Synchronizacja zadań – konstrukcja `barrier`

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            #pragma omp task
            {
                taskB();
            }

            taskA();
        }

        #pragma omp task
        {
            #pragma omp task
            {
                taskD();
            }

            taskC();
        }
    }

    #pragma omp barrier

    #pragma omp master
    {
        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zadania

Wątek główny tworzy dwa zadania

Każde z zadań tworzy zdania potomne

Wychodzimy z obszaru realizowanego przez wątek główny

Bariera musi zostać osiągnięta przez wszystkie uruchomione wątki

Synchronizacja zadań – konstrukcja `barrier`

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            #pragma omp task
            {
                taskB();
            }

            taskA();
        }

        #pragma omp task
        {
            #pragma omp task
            {
                taskD();
            }

            taskC();
        }
    }

    #pragma omp barrier

    #pragma omp master
    {
        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zadania

Wątek główny tworzy dwa zadania

Każde z zadań tworzy zdania potomne

Wychodzimy z obszaru realizowanego przez wątek główny

Bariera musi zostać osiągnięta przez wszystkie uruchomione wątki

Wszystkie utworzone zadania zakończą się w tym miejscu

Synchronizacja zadań – konstrukcja `barrier`

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            #pragma omp task
            {
                taskB();
            }

            taskA();
        }

        #pragma omp task
        {
            #pragma omp task
            {
                taskD();
            }

            taskC();
        }
    }

    #pragma omp barrier

    #pragma omp master
    {
        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zadania

Wątek główny tworzy dwa zadania

Każde z zadań tworzy zdania potomne

Wychodzimy z obszaru realizowanego przez wątek główny

Bariera musi zostać osiągnięta przez wszystkie uruchomione wątki

Wszystkie utworzone zadania zakończą się w tym miejscu

Zadanie `taskE` utworzone zostanie po zakończeniu wszystkich poprzednich zadań

Synchronizacja zadań za pomocą konstrukcji `taskgroup`

- Konstrukcja `taskgroup` podobnie do konstrukcji `taskwait` umożliwia synchronizację zadań
- Różnicą w porównaniu do `taskwait` jest fakt, iż konstrukcja `taskgroup` wymusza zakończenie wszystkich utworzonych zadań, łącznie z zadaniami potomnymi utworzonymi w ramach bloku, który poprzedza
- Ogólna postać konstrukcji `taskgroup` wygląda następująco:

```
#pragma omp taskgroup [klauzla]
```

Synchronizacja zadań za pomocą konstrukcji `taskgroup`

- Konstrukcja `taskgroup` posiada jedną klauzulę:
  - `task_reduction(reduction-identifier|list)`: redukcja obsługiwana tylko przez dyrektywę `taskgroup` w połączeniu z klauzulą `in_reduction` stosowaną dla zadań utworzonych za pomocą dyrektywy `#pragma omp task`. Po wykonaniu zadań w ramach bloku `taskgroup` zmienna na liście będzie posiadała nową wartość.

Synchronizacja zadań za pomocą konstrukcji `taskgroup`

- Przykład synchronizacji zadań z wykorzystaniem konstrukcji `takgroup`:

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp taskgroup
        {
            #pragma omp task
            {
                #pragma omp task
                {
                    taskD();
                }

                taskA();
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    taskC();
                }

                taskB();
            }
        }

        #pragma omp task
        {
            taskE();
        }
    }
}
```

Synchronizacja zadań za pomocą konstrukcji `taskgroup`

- Przykład synchronizacji zadań z wykorzystaniem konstrukcji `takgroup`:

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp taskgroup
        {
            #pragma omp task
            {
                #pragma omp task
                {
                    taskD();
                }

                taskA();
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    taskC();
                }

                taskB();
            }
        }

        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zdania

Synchronizacja zadań za pomocą konstrukcji `taskgroup`

- Przykład synchronizacji zadań z wykorzystaniem konstrukcji `takgroup`:

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp taskgroup
        {
            #pragma omp task
            {
                #pragma omp task
                {
                    taskD();
                }

                taskA();
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    taskC();
                }

                taskB();
            }
        }

        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zdania

Wszystkie zadania tworzone są w ramach konstrukcji `taskgroup`

Synchronizacja zadań za pomocą konstrukcji `taskgroup`

- Przykład synchronizacji zadań z wykorzystaniem konstrukcji `takgroup`:

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp taskgroup
        {
            #pragma omp task
            {
                #pragma omp task
                {
                    taskD();
                }

                taskA();
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    taskC();
                }

                taskB();
            }
        }

        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zdania

Wszystkie zadania tworzone są w ramach konstrukcji `taskgroup`

Zadania tworzone przez wątek główny tworzą zadania potomne `taskD` oraz `taskC`

Synchronizacja zadań za pomocą konstrukcji `taskgroup`

- Przykład synchronizacji zadań z wykorzystaniem konstrukcji `takgroup`:

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp taskgroup
        {
            #pragma omp task
            {
                #pragma omp task
                {
                    taskD();
                }

                taskA();
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    taskC();
                }

                taskB();
            }
        }

        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zdania

Wszystkie zadania tworzone są w ramach konstrukcji `taskgroup`

Zadania tworzone przez wątek główny tworzą zadania potomne `taskD` oraz `taskC`

Na końcu bloku `taskgroup` następuje synchronizacja wszystkich utworzonych wewnątrz zadań

Synchronizacja zadań za pomocą konstrukcji `taskgroup`

- Przykład synchronizacji zadań z wykorzystaniem konstrukcji `takgroup`:

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp taskgroup
        {
            #pragma omp task
            {
                #pragma omp task
                {
                    taskD();
                }

                taskA();
            }

            #pragma omp task
            {
                #pragma omp task
                {
                    taskC();
                }

                taskB();
            }
        }

        #pragma omp task
        {
            taskE();
        }
    }
}
```

Wątek główny tworzy zdania

Wszystkie zadania tworzone są w ramach konstrukcji `taskgroup`

Zadania tworzone przez wątek główny tworzą zadania potomne `taskD` oraz `taskC`

Na końcu bloku `taskgroup` następuje synchronizacja wszystkich utworzonych wewnątrz zadań

W rezultacie, zadanie `taskE` utworzone zostanie dopiero po wykonaniu zadań `taskA`, `taskB`, `taskC` oraz `taskD`

Przykład XI – konstrukcja taskgroup

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

```cpp
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład XI – konstrukcja taskgroup

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

```cpp
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład XI – konstrukcja taskgroup

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Tworzymy trzy tablice

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład XI – konstrukcja taskgroup

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Tworzymy trzy tablice

Zmienna max_avg jest współdzielona przez wątki

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład XI – konstrukcja taskgroup

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Tworzymy trzy tablice

Zmienna max_avg jest współdzielona przez wątki

Tworzymy trzy wątki

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład XI – konstrukcja taskgroup

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Tworzymy trzy tablice

Zmienna max_avg jest współdzielona przez wątki

Tworzymy trzy wątki

Wątek główny tworzy zdania

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład XI – konstrukcja `taskgroup`

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Tworzymy trzy tablice

Zmienna `max_avg` jest współdzielona przez wątki

Tworzymy trzy wątki

Wątek główny tworzy zdania

Zadania tworzone są w tamach konstrukcji `taskgroup`

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład XI – konstrukcja `taskgroup`

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Zmienna `max_avg` jest współdzielona przez wątki

Tworzymy trzy wątki

Wątek główny tworzy zdania

Zadania tworzone są w tamach konstrukcji `taskgroup`

Zadania tworzone są z wykorzystaniem klauzuli `in_reduction`

Przykład XI – konstrukcja `taskgroup`

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Tworzymy trzy tablice

Zmienna `max_avg` jest współdzielona przez wątki

Tworzymy trzy wątki

Wątek główny tworzy zdania

Zadania tworzone są w tamach konstrukcji `taskgroup`

Zadania tworzone są z wykorzystaniem klauzuli `in_reduction`

Dla każdej z tablicy wyznaczamy wartość średnią z elementów w osobnej sekcji

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład XI – konstrukcja `taskgroup`

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Za pomocą redukcji task_reduction obliczamy wartość maksymalną ze wszystkich średnich w ramach zadań, a następnie zapisujemy jej wartość do zmiennej współdzielonej `min_avg`

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Zmienna `max_avg` jest współdzielona przez wątki

Tworzymy trzy wątki

Wątek główny tworzy zdania

Zadania tworzone są w tamach konstrukcji `taskgroup`

Zadania tworzone są z wykorzystaniem klauzuli `in_reduction`

Dla każdej z tablicy wyznaczamy wartość średnią z elementów w osobnej sekcji

Przykład XI – konstrukcja `taskgroup`

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

Tworzymy trzy tablice

Zmienna `max_avg` jest współdzielona przez wątki

Tworzymy trzy wątki

Wątek główny tworzy zdania

Zadania tworzone są w tamach konstrukcji `taskgroup`

Zadania tworzone są z wykorzystaniem klauzuli `in_reduction`

Za pomocą redukcji task_reduction obliczamy wartość maksymalną ze wszystkich średnich w ramach zadań, a następnie zapisujemy jej wartość do zmiennej współdzielonej `min_avg`

Dla każdej z tablicy wyznaczamy wartość średnią z elementów w osobnej sekcji

Wyświetlamy największą wartość średnią

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
    for(int i=0; i<size; ++i)
    {
        A[i] = rand_double();
        B[i] = rand_double();
        C[i] = rand_double();
    }

    double max_avg = 0.0;
    #pragma omp parallel num_threads(3)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(max : max_avg)
            {
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(A, size);
                    string output = "Tablica A: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(B, size);
                    string output = "Tablica B: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
                #pragma omp task in_reduction(max : max_avg)
                {
                    max_avg = average(C, size);
                    string output = "Tablica C: srednia = " + to_string(max_avg) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Przykład XI – konstrukcja `taskgroup`

Funkcja obliczająca średnią z tablicy

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

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

    return sum / n;
}

double rand_double()
{
    double f = (double)rand() / RAND_MAX;
    return f + (double)(rand() % 1000);
}
```

```
int main()
{
    srand(time(NULL));
    int size = 100;
    double *A = new double[size];
    double *B = new double[size];
    double *C = new double[size];
```

Tworzymy trzy tablice

Zmienna `max_avg` jest współdzielona przez wątki

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++-9 -fopenmp ./t6_omp15.cpp
rid@gpu2:~/RID/T6$ ./a.out
Tablica C: srednia = 493.656678
Tablica A: srednia = 494.882299
Tablica B: srednia = 516.160201

Srednia max: 516.16
rid@gpu2:~/RID/T6$ ./a.out
Tablica C: srednia = 464.776668
Tablica B: srednia = 497.520872
Tablica A: srednia = 509.950627

Srednia max: 509.951
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

worzy zdania

rzone są w tamach kcji `taskgroup`

lania tworzone są wykorzystaniem zuli `in_reduction`

Za pomocą redukcji task_re obliczamy wartość maksy ze wszystkich średnich w zadań, a następnie zapisujemy jej wartość do z współdzielonej `min_avg`

wyznaczamy wartość średnią z elementów w osobnej sekcji

```
    cout << "\nSrednia max: " << max_avg << "\n";

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Wyświetlamy największą wartość średnią

Przykład XII – konstrukcja taskgroup

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}

int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task reduction(+ : x)
            {
                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Redukcja w ramach konstrukcji taskgroup nie musi zostać wykonana dla wszystkich zadań utworzonych wewnątrz !

Przykład XII – konstrukcja taskgroup

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}

int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(+ : x)
            {
                #pragma omp task in_reduction(+ : x)
                {
                    x += fun1();
                    string output = "Task A - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task in_reduction(+ : x)
                {
                    x += fun2();
                    string output = "Task B - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Przykład XII – konstrukcja taskgroup

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}
```

Funkcje wykonywane przez dwa zdania

```cpp
int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(+ : x)
            {
                #pragma omp task in_reduction(+ : x)
                {
                    x += fun1();
                    string output = "Task A - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task in_reduction(+ : x)
                {
                    x += fun2();
                    string output = "Task B - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Przykład XII – konstrukcja taskgroup

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}
```

Funkcje wykonywane przez dwa zdania

```cpp
int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(+ : x)
            {
                #pragma omp task in_reduction(+ : x)
                {
                    x += fun1();
                    string output = "Task A - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task in_reduction(+ : x)
                {
                    x += fun2();
                    string output = "Task B - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Zmienna x jest współdzielona przez wątki

Przykład XII – konstrukcja taskgroup

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}
```

Funkcje wykonywane przez dwa zdania

```cpp
int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(+ : x)
            {
                #pragma omp task in_reduction(+ : x)
                {
                    x += fun1();
                    string output = "Task A - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task in_reduction(+ : x)
                {
                    x += fun2();
                    string output = "Task B - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Zmienna x jest współdzielona przez wątki

Wątek główny tworzy zadania

Przykład XII – konstrukcja taskgroup

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}
```

Funkcje wykonywane przez dwa zdania

```cpp
int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(+ : x)
            {
                #pragma omp task in_reduction(+ : x)
                {
                    x += fun1();
                    string output = "Task A - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task in_reduction(+ : x)
                {
                    x += fun2();
                    string output = "Task B - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Zmienna x jest współdzielona przez wątki

Wątek główny tworzy zadania

Zadania tworzone są w ramach konstrukcji taskgroup

Przykład XII – konstrukcja `taskgroup`

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}
```

Funkcje wykonywane przez dwa zdania

```cpp
int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(+ : x)
            {
                #pragma omp task in_reduction(+ : x)
                {
                    x += fun1();
                    string output = "Task A - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task in_reduction(+ : x)
                {
                    x += fun2();
                    string output = "Task B - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Zmienna x jest współdzielona przez wątki

Wątek główny tworzy zadania

Zadania tworzone są w ramach konstrukcji `taskgroup`

Wartość zmiennej x jest redukowana po wykonaniu zadań

Przykład XII – konstrukcja `taskgroup`

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}
```

Funkcje wykonywane przez dwa zdania

```cpp
int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(+ : x)
            {
                #pragma omp task in_reduction(+ : x)
                {
                    x += fun1();
                    string output = "Task A - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task in_reduction(+ : x)
                {
                    x += fun2();
                    string output = "Task B - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Zmienna x jest współdzielona przez wątki

Wątek główny tworzy zadania

Zadania tworzone są w ramach konstrukcji `taskgroup`

Wartość zmiennej x jest redukowana po wykonaniu zadań

Dwa zadania utworzone zostały z wykorzystaniem klauzuli `in_reducion` na rzecz `x`

Przykład XII – konstrukcja `taskgroup`

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}
```

Funkcje wykonywane przez dwa zdania

```cpp
int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(+ : x)
            {
                #pragma omp task in_reduction(+ : x)
                {
                    x += fun1();
                    string output = "Task A - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task in_reduction(+ : x)
                {
                    x += fun2();
                    string output = "Task B - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Zmienna x jest współdzielona przez wątki

Wątek główny tworzy zadania

Zadania tworzone są w ramach konstrukcji `taskgroup`

Wartość zmiennej x jest redukowana po wykonaniu zadań

Dwa zadania utworzone zostały z wykorzystaniem klauzuli `in_reducion` na rzecz `x`

Trzecie zadanie nie korzysta z redukcji. Zmienna x jest w tym przypadku `firstprivate`

Przykład XII – konstrukcja `taskgroup`

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}
```

Funkcje wykonywane przez dwa zdania

```cpp
int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp master
        {
            #pragma omp taskgroup task_reduction(+ : x)
            {
                #pragma omp task in_reduction(+ : x)
                {
                    x += fun1();
                    string output = "Task A - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task in_reduction(+ : x)
                {
                    x += fun2();
                    string output = "Task B - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }

                #pragma omp task firstprivate(x)
                {
                    sleep(1);
                    x += rand() % 100;
                    string output = "Task C - wartosc x=" + to_string(x) + "\n";
                    cout << output;
                }
            }
        }
    }
    cout << "Wartosc zmiennej x po redukcji: " << x << "\n";
    return 0;
}
```

Zmienna x jest współdzielona przez wątki

Wątek główny tworzy zadania

Zadania tworzone są w ramach konstrukcji `taskgroup`

Wartość zmiennej x jest redukowana po wykonaniu zadań

Dwa zadania utworzone zostały z wykorzystaniem klauzuli `in_reducion` na rzecz `x`

Trzecie zadanie nie korzysta z redukcji. Zmienna x jest w tym przypadku `firstprivate`

Po wykonaniu bloku równoległego wyświetlamy wartość zmiennej x

Przykład XII – konstrukcja taskgroup

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

int fun1()
{
    sleep(1);
    return 10;
}

int fun2()
{
    sleep(1);
    return 20;
}
```

```
int main()
{
    srand(time(0));
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragm
        {
            #p
            {
            }
        }
    }
    cout << "W
    return 0;
}
```

Zmienna x jest współdzielona przez wątki

Watek główny tworzy zadania

Funkcje wykonywane przez dwa zdania

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++-9 -fopenmp ./t6_omp16.cpp
rid@gpu2:~/RID/T6$ ./a.out
Task A - wartosc x=10
Task B - wartosc x=20
Task C - wartosc x=33

Wartosc zmiennej x po redukcji: 30
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Redukcja wykonana została jednie dla dwóch zadań

ach

x jest
aniu zadań

zone zostały
n klauzuli
a rzecz x

korzysta
a x jest
stprivate

noległego
miennej x

Konstrukcja `taskyield`

- Konstrukcja `taskyield` określa, że bieżące zadanie może zostać zawieszone na korzyść wykonania innego zadania
- Dyrektywa ta stanowi wskazówkę dla środowiska wykonawczego OpenMP i ma na celu optymalizację i / lub zapobieganie zakleszczeniom
- Ogólna postać dyrektywy taskyield wygląda następująco:
  `#pragma omp taksyield`
- Konstrukcja ta wywoływana jest w ramach zadania

Przykład XIII – konstrukcja `taskyield`

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return 0;
}
```

Przykład XIII – konstrukcja `taskyield`

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return 0;
}
```

Przykład XIII – konstrukcja `taskyield`

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)

Inicjalizujemy zamek

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return 0;
}
```

Przykład XIII – konstrukcja `taskyield`

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)

Inicjalizujemy zamek

Tworzymy cztery wątki

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return 0;
}
```

Przykład XIII – konstrukcja `taskyield`

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return 0;
}
```

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)

Inicjalizujemy zamek

Tworzymy cztery wątki

Wątki tworzą zadanie, które po wznowieniu może zostać wykonane przez inny wątek

Przykład XIII – konstrukcja `taskyield`

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return 0;
}
```

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)

Inicjalizujemy zamek

Tworzymy cztery wątki

Wątki tworzą zadanie, które po wznowieniu może zostać wykonane przez inny wątek

Wątek sprawdza czy może założyć blokadę

Przykład XIII – konstrukcja `taskyield`

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return 0;
}
```

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)

Inicjalizujemy zamek

Tworzymy cztery wątki

Wątki tworzą zadanie, które po wznowieniu może zostać wykonane przez inny wątek

Wątek sprawdza czy może założyć blokadę

Jeśli założenie blokady nie jest możliwe zawieszamy zadanie

Przykład XIII – konstrukcja `taskyield`

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return 0;
}
```

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)

Inicjalizujemy zamek

Tworzymy cztery wątki

Wątki tworzą zadanie, które po wznowieniu może zostać wykonane przez inny wątek

Wątek sprawdza czy może założyć blokadę

Jeśli założenie blokady nie jest możliwe zawieszamy zadanie

Jeśli możliwe jest założenie blokady, ustawiamy ją i sypiamy wątek

Przykład XIII – konstrukcja `taskyield`

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return 0;
}
```

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)

Inicjalizujemy zamek

Tworzymy cztery wątki

Wątki tworzą zadanie, które po wznowieniu może zostać wykonane przez inny wątek

Wątek sprawdza czy może założyć blokadę

Jeśli założenie blokady nie jest możliwe zawieszamy zadanie

Jeśli możliwe jest założenie blokady, ustawiamy ją i sypiamy wątek

Po wybudzeniu wątku, zwalniamy zamek

Przykład XIII – konstrukcja `taskyield`

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4)
    {
        #pragma omp task untied
        {
            cout << "Watek #" + to_string(omp_get_thread_num()) + " czeka na wykonanie lock\n";
            while ( !omp_test_lock(&lock ) )
            {
                #pragma omp taskyield
            }

            cout << "Watek #" + to_string(omp_get_thread_num()) + " jest uspiony\n";
            sleep(1);
            cout << "Watek #" + to_string(omp_get_thread_num()) + " wykonal lock\n";
            omp_unset_lock(&lock ) ;
        }
    }

    omp_destroy_lock(&lock ) ;

    return
}
```

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)

Inicjalizujemy zamek

Tworzymy cztery wątki

Wątki tworzą zadanie, które po wznowieniu może zostać wykonane przez inny wątek

Wątek sprawdza czy może założyć blokadę

Jeśli założenie blokady nie jest możliwe zawieszamy zadanie

Jeśli możliwe jest założenie blokady, ustawiamy ją i sypiamy wątek

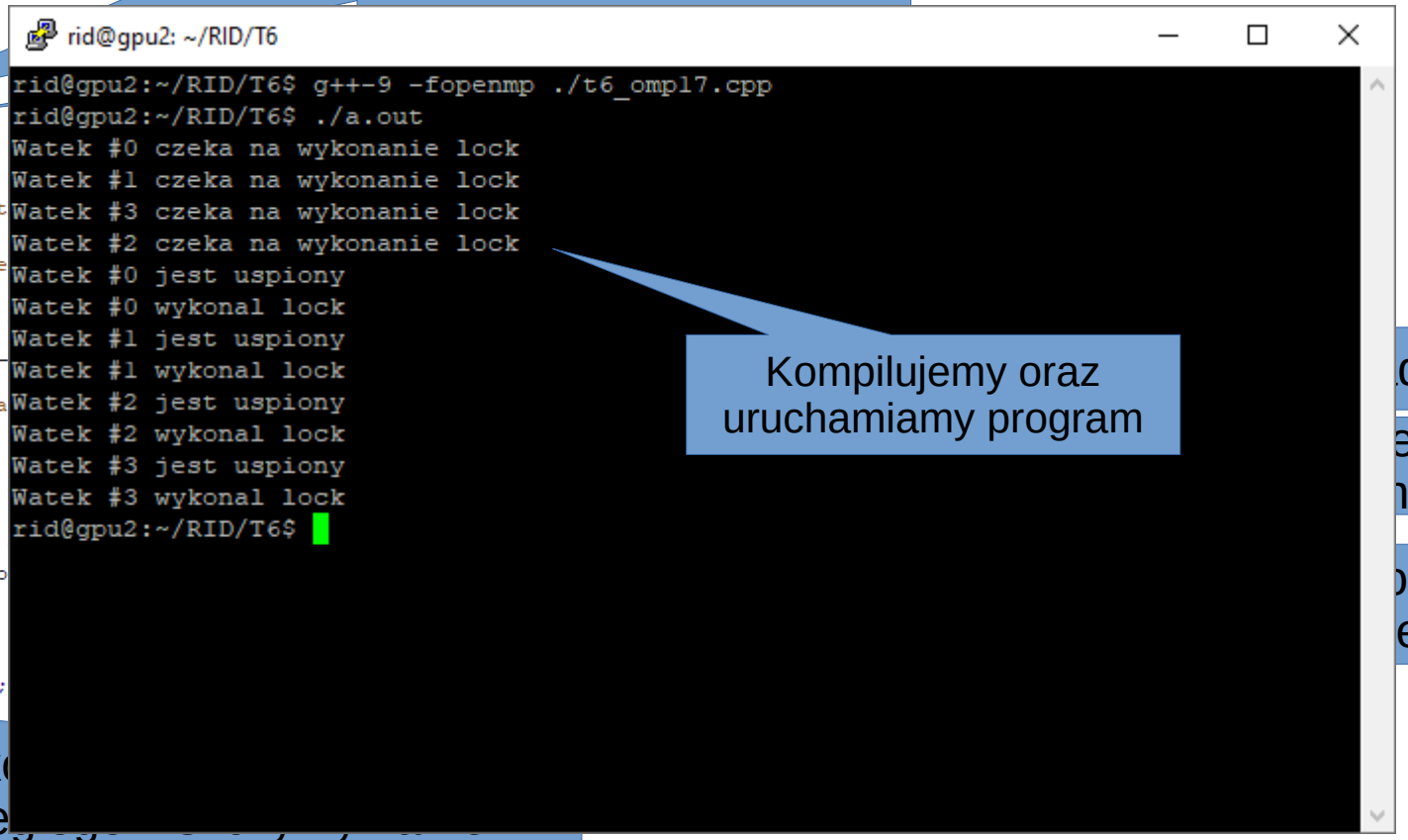
Po wybudzeniu wątku, zwalniamy zamek

Po zakończeniu przetwarzania równoległego niszczymy zamek

Przykład XIII – konstrukcja `taskyield`

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

int main()
{
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel num_t
    {
        #pragma omp task untie
        {
            cout << "Watek #"
            while ( !omp_test_
            {
                #pragma omp ta
            }

            cout << "Watek #"
            sleep(1);
            cout << "Watek #"
            omp_unset_lock(&lo
        }
    }

    omp_destroy_lock(&lock ) ;

    return
}
```

Zamek (ang. *lock*) jest służy do zapobiegania konfliktom w dostępnie do zasobów (sekcja krytyczna)



Kompilujemy oraz uruchamiamy program

Po zak
równole

dę
est
nie
kady,
ek

Zrównoleglenie pętli przy użyciu zadań

- Standard OpenMP dostarcza konstrukcje `taskloop` pozwalającą na zrównoleglenie pętli `for` z wykorzystaniem zadań
- Ogólna postać dyrektywy `taskloop`:
  `#pragma omp taskloop [klazula1, [klauzula2] …]`
- Jak sama nazwa wskazuje konstrukcja `taskloop` przekształca iteracje pętli w zadania
  - Pętla dzielona jest na części, następnie dla każdej z części tworzone jest zadanie
- Zadania wykonywane są niejawnie w ramach bloku `taskgroup`
- Konstrukcja ta nie jest konstrukcją współdzielenia pracy (jak `omp for`), należy ją wywołać w bloku `single` lub `master`, inaczej pętla wykonana zostanie wiele razy
- Konstrukcja `taskloop` posiada także drugi wariant `taskloop simd`, który określa, że pętla ma zostać wykonana z wykorzystaniem instrukcji SIMD, i że iteracje mają być wykonywane równolegle przy użyciu zadań OpenMP

Zrównoleglenie pętli przy użyciu zadań

- Klauzule dyrektywy `#pragma omp taskloop`:
  - Klauzule widoczności zmiennych: `shared`, `private`, `firstprivate`, `lastprivate`, `default(shared|none)`
  - Klauzule konstrukcji `task`: `final`, `untied`, `mergable`, `priority`, `if`
  - Klauzula `reduction(operator | intrinsic: list)`
  - `collapse(n)`: określa, ile pętli w zagnieżdżonej pętli należy zwinąć w jedną dużą przestrzeń iteracyjną i podzielić zgodnie z klauzulą harmonogramu

Zrównoleglenie pętli przy użyciu zadań

- Klauzule dyrektywy `#pragma omp taskloop`:
  - `grainsize(value)`: określa liczbę iteracji pętli przypisanych do każdego utworzonego zadania. Przekazywana wartość musi być dodatnia
  - `num_tasks(value)`: Określa ilość zadań utworzonych w trakcie wykonania pętli. Przekazywana wartość musi być dodatnia.
  - `nogroup`: usuwa niejawny region `taskgroup`
- Konstrukcja `taskloop simd`, która zrównolegla pętle oraz wektoryzuje wykonywane w niej obliczenia dziedziczy klauzule dyrektyw `#pragma omp taskloop` oraz `#pragma omp simd`

Zrównoleglenie pętli przy użyciu zadań

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

int main()
{
    srand(time(0));
    const int size = 100;
    int *A = new int[size];
    int *B = new int[size];
    int *C = new int[size];

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

    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskloop grainsize(20)
            for(int i=0; i<size; ++i)
            {
                C[i] = A[i] + B[i];
            }
        }
    }

    //...

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Zrównoleglenie pętli przy użyciu zadań

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

int main()
{
    srand(time(0));
    const int size = 100;
    int *A = new int[size];
    int *B = new int[size];
    int *C = new int[size];

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

    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskloop grainsize(20)
            for(int i=0; i<size; ++i)
            {
                C[i] = A[i] + B[i];
            }
        }
    }

    //...

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Zrównoleglenie pętli przy użyciu zadań

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

int main()
{
    srand(time(0));
    const int size = 100;
    int *A = new int[size];
    int *B = new int[size];
    int *C = new int[size];

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

    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskloop grainsize(20)
            for(int i=0; i<size; ++i)
            {
                C[i] = A[i] + B[i];
            }
        }
    }

    //...

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Wypełniamy tablice wartościami losowymi

Zrównoleglenie pętli przy użyciu zadań

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

int main()
{
    srand(time(0));
    const int size = 100;
    int *A = new int[size];
    int *B = new int[size];
    int *C = new int[size];

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

    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskloop grainsize(20)
            for(int i=0; i<size; ++i)
            {
                C[i] = A[i] + B[i];
            }
        }
    }

    //...

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Wypełniamy tablice wartościami losowymi

Tworzymy obszar równoległy

Zrównoleglenie pętli przy użyciu zadań

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

int main()
{
    srand(time(0));
    const int size = 100;
    int *A = new int[size];
    int *B = new int[size];
    int *C = new int[size];

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

    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskloop grainsize(20)
            for(int i=0; i<size; ++i)
            {
                C[i] = A[i] + B[i];
            }
        }
    }

    //...

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Wypełniamy tablice wartościami losowymi

Tworzymy obszar równoległy

Zrównoleglamy pętle w wykorzystaniem konstrukcji `taskloop`

Zrównoleglenie pętli przy użyciu zadań

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

int main()
{
    srand(time(0));
    const int size = 100;
    int *A = new int[size];
    int *B = new int[size];
    int *C = new int[size];

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

    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskloop grainsize(20)
            for(int i=0; i<size; ++i)
            {
                C[i] = A[i] + B[i];
            }
        }
    }

    //...

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Wypełniamy tablice wartościami losowymi

Tworzymy obszar równoległy

Zrównoleglamy pętle w wykorzystaniem konstrukcji `taskloop`

Klauzula `grainsize` określa ile iteracji pętli wykonane zostanie w ramach zadania

Zrównoleglenie pętli przy użyciu zadań

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

int main()
{
    srand(time(0));
    const int size = 100;
    int *A = new int[size];
    int *B = new int[size];
    int *C = new int[size];

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

    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskloop grainsize(20)
            for(int i=0; i<size; ++i)
            {
                C[i] = A[i] + B[i];
            }
        }
    }

    //...

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Wypełniamy tablice wartościami losowymi

Tworzymy obszar równoległy

Zrównoleglamy pętle w wykorzystaniem konstrukcji `taskloop`

Klauzula `grainsize` określa ile iteracji pętli wykonane zostanie w ramach zadania

Każde zadanie wykona 20 iteracji pętli

Zrównoleglenie pętli przy użyciu zadań

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

int main()
{
    srand(time(0));
    const int size = 100;
    int *A = new int[size];
    int *B = new int[size];
    int *C = new int[size];

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

    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskloop grainsize(20)
            for(int i=0; i<size; ++i)
            {
                C[i] = A[i] + B[i];
            }
        }
    }

    //...

    delete[] C;
    delete[] B;
    delete[] A;
    return 0;
}
```

Tworzymy trzy tablice

Wypełniamy tablice wartościami losowymi

Tworzymy obszar równoległy

Zrównoleglamy pętle w wykorzystaniem konstrukcji `taskloop`

Klauzula `grainsize` określa ile iteracji pętli wykonane zostanie w ramach zadania

Każde zadanie wykona 20 iteracji pętli

Konstrukcja `taskloop` zastosowana została w ramach bloku `master`

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

```
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

Wersja sekwencyjna algorytmu

```cpp
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

Wersja sekwencyjna algorytmu

Kod równoległy

```cpp
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

Wersja sekwencyjna algorytmu

Kod równoległy

Tworzymy cztery wątki

```cpp
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

Wersja sekwencyjna algorytmu

Kod równoległy

Tworzymy cztery wątki

Iteracje pętli realizowane są przez cztery zadania

```cpp
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}

int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Wersja sekwencyjna algorytmu

Kod równoległy

Tworzymy cztery wątki

Iteracje pętli realizowane są przez cztery zadania

Pętle zrównoleglamy z wykorzystaniem konstrukcji `taskloop`

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

Wersja sekwencyjna algorytmu

Kod równoległy

Tworzymy cztery wątki

Iteracje pętli realizowane są przez cztery zadania

Wykonujemy redukcję na zmiennej współdzielonej `sum`

Pętle zrównoleglamy z wykorzystaniem konstrukcji `taskloop`

```cpp
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

Wersja sekwencyjna algorytmu

Kod równoległy

Tworzymy cztery wątki

Iteracje pętli realizowane są przez cztery zadania

Wykonujemy redukcję na zmiennej współdzielonej `sum`

Pętle zrównoleglamy z wykorzystaniem konstrukcji `taskloop`

```cpp
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Dokładność przybliżenia liczby PI

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

Wersja sekwencyjna algorytmu

Kod równoległy

Tworzymy cztery wątki

Iteracje pętli realizowane są przez cztery zadania

Wykonujemy redukcję na zmiennej współdzielonej `sum`

Pętle zrównoleglamy z wykorzystaniem konstrukcji `taskloop`

```cpp
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Dokładność przybliżenia liczby PI

Wersja sekwencyjna

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

Wersja sekwencyjna algorytmu

Kod równoległy

Tworzymy cztery wątki

Iteracje pętli realizowane są przez cztery zadania

Wykonujemy redukcję na zmiennej współdzielonej `sum`

Pętle zrównoleglamy z wykorzystaniem konstrukcji `taskloop`

```cpp
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Dokładność przybliżenia liczby PI

Wersja sekwencyjna

Wersja równoległa

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx) private(xi) reduction(+ : sum) num_tasks(4)
            for (long int i=0; i<steps; ++i)
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*xi);
            }
        }
    }

    pi = (dx * sum);
    return pi;
}
```

```cpp
int main()
{
    double start, stop;
    double pi_s, pi_p;
    const int steps = 500000000;

    cout << "Kod sekwencyjny...\n";
    start = omp_get_wtime();
    pi_s = pi_serial(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_s << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    cout << "\nKod rownolegly...\n";
    start = omp_get_wtime();
    pi_p = pi_parallel(steps);
    stop = omp_get_wtime();
    cout << "PI = " << pi_p << "\n";
    cout << "Czas obliczen: " << stop-start << endl;

    return 0;
}
```

Wersja sekwencyjna algorytmu

Kod równoległy

Tworzymy cztery wątki

Iteracje pętli realizowane są przez cztery zadania

Wykonujemy redukcję na zmiennej współdzielonej `sum`

Pętle zrównoleglamy z wykorzystaniem konstrukcji `taskloop`

Dokładność przybliżenia liczby PI

Wersja sekwencyjna

Wersja równoległa

Mierzymy czas obliczeń obu wersji programu

Przykład XIV – taskloop reduction

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

double pi_serial(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    for (long int i=0; i<steps; ++i)
    {
        xi = ((double)i+0.5)*dx;
        sum = sum + 4.0 / (1.0+xi*xi);
    }

    pi = (dx * sum);
    return pi;
}

double pi_parallel(const int steps)
{
    double pi, xi;
    double sum = 0.0;
    const double dx = 1.0/(double)steps;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            #pragma omp taskloop shared(dx)
            for (long int i=0; i<steps; ++i
            {
                xi = ((double)i+0.5)*dx;
                sum = sum + 4.0 / (1.0+xi*x
            }
        }
    }
    pi = (dx * sum);
    return pi;
}
```

```
int main()
{
    double start, stop;
```

Wersja sekwencyjna algorytmu

Kod ró

T

Pęt
wykorzystaniem konstrukcji `taskloop`

Dokładność przybliżenia liczby PI

Wersja sekwencyjna

ja równoległa

s obliczeń
rogramu

```
rid@gpu2: ~/RID/T6
rid@gpu2:~/RID/T6$ g++-9 -O3 -fopenmp ./t6_omp19.cpp
rid@gpu2:~/RID/T6$ ./a.out
Kod sekwencyjny...
PI = 3.14159
Czas obliczen: 2.00216

Kod rownolegly...
PI = 3.14159
Czas obliczen: 0.617972
rid@gpu2:~/RID/T6$
```

Kompilujemy oraz uruchamiamy program

Zrównoleglenie pętli pozwoliło skrócić czas obliczeń

Przykład XV – taskloop nogroup

```
void parallel_work()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    long_taskA();
                }

                #pragma omp taskloop grainsize(500) nogroup
                for(int i=0; i<10000; ++i)
                    for(int j=0; j<500; ++j)
                        loop_body(i,j)
            }

            #pragma omp task
            {
                long_taskB();
            }
        }
    }
}
```

Przykład XV – taskloop nogroup

Obszar równoległy

```
void parallel_work()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    long_taskA();
                }

                #pragma omp taskloop grainsize(500) nogroup
                for(int i=0; i<10000; ++i)
                    for(int j=0; j<500; ++j)
                        loop_body(i,j)
            }

            #pragma omp task
            {
                long_taskB();
            }
        }
    }
}
```

Przykład XV – taskloop nogroup

Obszar równoległy

Wątek główny tworzy zadania

```
void parallel_work()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    long_taskA();
                }

                #pragma omp taskloop grainsize(500) nogroup
                for(int i=0; i<10000; ++i)
                    for(int j=0; j<500; ++j)
                        loop_body(i,j)
            }

            #pragma omp task
            {
                long_taskB();
            }
        }
    }
}
```

Przykład XV – taskloop nogroup

Obszar równoległy

Wątek główny tworzy zadania

Zadania tworzone są w ramach bloku taskgroup

```
void parallel_work()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    long_taskA();
                }

                #pragma omp taskloop grainsize(500) nogroup
                for(int i=0; i<10000; ++i)
                    for(int j=0; j<500; ++j)
                        loop_body(i,j)
            }

            #pragma omp task
            {
                long_taskB();
            }
        }
    }
}
```

Przykład XV – taskloop nogroup

```
void parallel_work()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    long_taskA();
                }

                #pragma omp taskloop grainsize(500) nogroup
                for(int i=0; i<10000; ++i)
                    for(int j=0; j<500; ++j)
                        loop_body(i,j)
            }

            #pragma omp task
            {
                long_taskB();
            }
        }
    }
}
```

Obszar równoległy

Wątek główny tworzy zadania

Zadania tworzone są w ramach bloku `taskgroup`

Na początku tworzone jest zadanie `long_taskA()`

Przykład XV – `taskloop nogroup`

```
void parallel_work()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    long_taskA();
                }

                #pragma omp taskloop grainsize(500) nogroup
                for(int i=0; i<10000; ++i)
                    for(int j=0; j<500; ++j)
                        loop_body(i,j)
            }

            #pragma omp task
            {
                long_taskB();
            }
        }
    }
}
```

Obszar równoległy

Wątek główny tworzy zadania

Zadania tworzone są w ramach bloku `taskgroup`

Na początku tworzone jest zadanie `long_taskA()`

Następnie zrównoleglamy pętle z wykorzystaniem konstrukcji `taskloop`

Przykład XV – taskloop nogroup

```
void parallel_work()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    long_taskA();
                }

                #pragma omp taskloop grainsize(500) nogroup
                for(int i=0; i<10000; ++i)
                    for(int j=0; j<500; ++j)
                        loop_body(i,j)
            }

            #pragma omp task
            {
                long_taskB();
            }
        }
    }
}
```

Obszar równoległy

Wątek główny tworzy zadania

Zadania tworzone są w ramach bloku `taskgroup`

Na początku tworzone jest zadanie `long_taskA()`

Następnie zrównoleglamy pętle z wykorzystaniem konstrukcji `taskloop`

Usuwamy niejawny `taskgroup`

Przykład XV – taskloop nogroup

```
void parallel_work()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    long_taskA();
                }

                #pragma omp taskloop grainsize(500) nogroup
                for(int i=0; i<10000; ++i)
                    for(int j=0; j<500; ++j)
                        loop_body(i,j)
            }

            #pragma omp task
            {
                long_taskB();
            }
        }
    }
}
```

Obszar równoległy

Wątek główny tworzy zadania

Zadania tworzone są w ramach bloku `taskgroup`

Na początku tworzone jest zadanie `long_taskA()`

Następnie zrównoleglamy pętle z wykorzystaniem konstrukcji `taskloop`

Usuwamy niejawny `taskgroup`

Pętla oraz zadanie `long_taskA()` realizowane są jednocześnie w ramach tego samego bloku `taskgroup`

Przykład XV – taskloop nogroup

```
void parallel_work()
{
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                #pragma omp task
                {
                    long_taskA();
                }

                #pragma omp taskloop grainsize(500) nogroup
                for(int i=0; i<10000; ++i)
                    for(int j=0; j<500; ++j)
                        loop_body(i,j)
            }

            #pragma omp task
            {
                long_taskB();
            }
        }
    }
}
```

Obszar równoległy

Wątek główny tworzy zadania

Zadania tworzone są w ramach bloku `taskgroup`

Na początku tworzone jest zadanie `long_taskA()`

Następnie zrównoleglamy pętle z wykorzystaniem konstrukcji `taskloop`

Usuwamy niejawny `taskgroup`

Pętla oraz zadanie `long_taskA()` realizowane są jednocześnie w ramach tego samego bloku `taskgroup`

Zadanie `long_taskB()` rozpocznie się po dopiero zakończeniu obliczeń w pętli oraz zadania `long_taskA()`

Zadanie 1

Napisać program operujący na tablicach jednowymiarowych, który realizuje następującą funkcjonalność:

- znajduje minimalny oraz maksymalny element tablicy;
- wyznacza sumę elementów tablic;
- wyznacza średnią wartość elementów tablic
- wyznacza iloczyn skalarny obu tablic.

Wszystkie wyżej wymienione operacje należy zrównoleglić z wykorzystaniem mechanizmu sekcji.

Zadanie 2

Zmodyfikować kod listy z dowiązaniami, tak aby pojedynczy element przechowywał dwie zmienne – a oraz b, reprezentujące boki prostokąta. Wypełnić listę wartościami losowymi. Zrównoleglić pętlę iterującą po elementach listy z wykorzystaniem zadań. Korzystając z mechanizmu redukcji obliczyć sumę pól wszystkich prostokątów. Wartości a oraz b wypełnić liczbami losowymi.

Zadanie 3

Zaimplementować równoległą wersję algorytmu wyznaczającego iloczyn skalarny dwóch wektorów z wykorzystaniem konstrukcji `#pragma omp taskloop`.

Dziękuję za uwagę!