

Wykład OpenMP część 1

Programowanie równoległe w systemie z pamięcią wspólną.

1. Podstawy OpenMP
2. Model fork-join
3. dyrektywy
4. klauzule

OpenMP - specyfikacja **dyrektyw kompilatora, bibliotek i zmiennych środowiskowych** na potrzeby przetwarzania współbieżnego w Fortranie i C/C++ opartego o mechanizm pamięci współdzielonej oraz wątki.

OpenMP został zaprojektowany przez głównych twórców sprzętu i oprogramowania zapewnia przenośne i skalowalne wspólne API dla różnych platform sprzętowych i programowych.

Ideą OpenMP jest stworzenie prostego mechanizmu wspierającego programowanie wielowątkowe, bez konieczności posiadania szczegółowej wiedzy o tworzeniu, synchronizacji i niszczeniu wątków. Zamiast tego programista otrzymuje możliwość bezpośredniego określania, które partie programu i jak mają być wykonane.

OpenMP

Zestaw kompilatorów AD 2009

Vendor/Source	Compiler	Information
»GNU	gcc (4.3.2)	Free and open source - Linux, Solaris, AIX, MacOSX, Windows Compile with -fopenmp »More information
»IBM	XL C/C++ / Fortran	Windows, AIX and Linux. http://www.ibm.com/software/rational/cafe/community/ccpp - the IBM C/C++ Community http://www-01.ibm.com/software/awdtools/fortran/ for XL Fortran http://www-01.ibm.com/software/awdtools/xlcpp/ for XL C/C++. »More information
»Sun Microsystems	C/C++ / Fortran	Sun Studio compilers and tools - free download for Solaris and Linux. Compile with -xopenmp »More information »Tools: Thread Analyzer/Performance Analyzer
»Intel	C/C++ / Fortran (10.1)	Windows, Linux, and MacOSX. Compile with -Qopenmp on Windows, or just -openmp on Linux or Mac OSX »More information
»Portland Group Compilers and Tools	C/C++ / Fortran	»More Information on PGI Compilers Compile with -mp »More Information on PGI OpenMP
»Absoft Pro FortranMP	Fortran	
»Lahey/Fujitsu Fortran 95	C/C++ / Fortran	»More Information
»PathScale	C/C++ / Fortran	Linux 32/64 bit. »PathScale Compiler Suite User Guide
»HP	C/C++ / Fortran	»More Information
»MS	Visual Studio 2008 C++	Implements OpenMP 2.0 »More Information

OpenMP

Zestaw kompilatorów AD 2011

Vendor/Source	Compiler	Information
» GNU	gcc (4.3.2)	Free and open source - Linux, Solaris, AIX, MacOSX, Windows Compile with <code>-fopenmp</code> More information
» IBM	XL C/C++ / Fortran	AIX and Linux. http://www.ibm.com/developerworks/rational/community/cafe/ccpp.html - the IBM C/C++ Community http://www-01.ibm.com/software/awdtools/fortran/ for XL Fortran http://www-01.ibm.com/software/awdtools/xlcpp/ for XL C/C++. More Information
» Oracle	C/C++ / Fortran	Oracle Solaris Studio compilers and tools - free download for Solaris and Linux. Compile with <code>-xopenmp</code> More information OpenMP User's Guide Tools: Thread Analyzer/Performance Analyzer
» Intel	C/C++ / Fortran (10.1)	Windows, Linux, and Mac OSX. Compile with <code>-Qopenmp</code> on Windows, or just <code>-openmp</code> on Linux or Mac OSX More information
» Portland Group Compilers and Tools	C/C++ / Fortran	More Information on PGI Compilers Compile with <code>-mp</code> More Information on PGI OpenMP
» Absoft Pro Fortran	Fortran	Version 11.1 of the Fortran 95 compiler for Linux, Windows and Mac OS X includes integrated OpenMP 3.0 support. Compile with <code>-openmp</code> . More Information
» Lahey/Fujitsu Fortran 95	C/C++ / Fortran	More Information
» PathScale	C/C++ / Fortran	Linux 32/64 bit. PathScale Compiler Suite User Guide
» HP	C/C++ / Fortran	More Information
» MS	Visual Studio 2008 C++	Implements OpenMP 2.0 More Information
» Cray	Cray C/C++ and Fortran	Supports OpenMP 3.0 on the Cray XT series Linux environment. OpenMP is on by default. More Information

OpenMP. Zestaw kompilatorów AD 2015

- **GNU** (Linux, Solaris, AIX, MacOSX, Windows, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, HPUX, RTEMS)
- IBM
- Oracle
- Intel
- Portland Group Compilers and Tools
- Absoft Pro Fortran
- Lahey/Fujitsu Fortran 95
- PathScale
- MS (Visual Studio 2008-2010 C++)
- Cray
- OpenUH Research Compiler
- NAG (nagfor)
- LLVM
- LLNL
- Texas Instruments
- BSC Mercurium
- Appentra Solutions parallelware compiler

OpenMP. Specyfikacje

<http://openmp.org/wp/openmp-specifications/>

- **2018.11 – wersja 5.0**
- 2015.11 – wersja 4.5
- 2013.06 – wersja 4.0
- 2011.06 – wersja 3.1
- 2008.05 – wersja 3.0
- 2005.05 – wersja 2.5
- 2002.03 – wersja 2.0
- 1998.10 – wersja 1.0

Wsparcie:

- gcc w wersji 4.7 – wsparcie v3.1, 4.9.1 – v4.0, 6.1 – v4.5
- MS Visual C++ – tylko na poziomie v2.0
- OpenUH – na poziomie v3.0
- Intel – v3.1 (v4.0 Intel C++ Composer XE 2013)

Internal parallelism maszyny – to kompilator musi zapewnić przystosowanie kodu do maszyny o wielu procesorach czy rdzeniach.

Tylko jak to zrobić (efektywnie) ? Już próbowało...

OpenMP udostępnia zestaw dyrektyw, którymi można poinstruować kompilator jak radzić sobie z naszym kodem.

Dyrektwy OpenMP można dodawać stopniowo przyspieszając i testując kod fragmentami. Kod nadal można uruchomić w wersji sekwencyjnej. Wystarczy więc JEDNA wersja kodu.

Kompilator nie jest zobowiązany do sprawdzenia poprawności kodu. Kod po zrównolegleniu może działać błędnie: produkować błędne rezultaty czy zakleszczać się.



W celu rozwiania wątpliwości:

- OpenMP nie zmodyfikuje za nas kodu źródłowego.
- To nie jest automatyczne zrównoleglanie algorytmu.
- OpenMP zrobi tylko to, co mu każe za pomocą dyrektyw programista.
- OpenMP nie odpowiada za błędy.
Używasz na własne ryzyko!



Wynik pracy programu może zależeć od ilości użytych wątków i w szczególności może być błędny !

OpenMP używa wątków

W OpenMP wiele wątków współdziała w celu realizacji programu

Wątki współdzielą zasoby. Mogą mieć także pewne zasoby na własność (prywatne).

Wątki mogą działać na wielu rzeczywistych procesorach (rdzeniach), ale program można uruchomić nawet na jednym.

Wątki realizują zadania (tasks)

Zadanie (task) – określona instancja wykonywalnego kodu + środowisko danych (data environment) wygenerowana gdy wątek napotyka konstrukcję **task** lub **parallel**

Zadania mogą być generowane jawnie lub niejawnie (automatycznie)

W jaki sposób program jest uruchamiany? Część sekwencyjna programu (cały program właściwie) objęta jest niejawnym i nieaktywnym obszarem równoległym (nieaktywny tzn., że jest wykonywany przez jeden wątek). To prowadzi do pojawienia się początkowego wątku (initial thread) i początkowego niejawnego zadania (initial task).

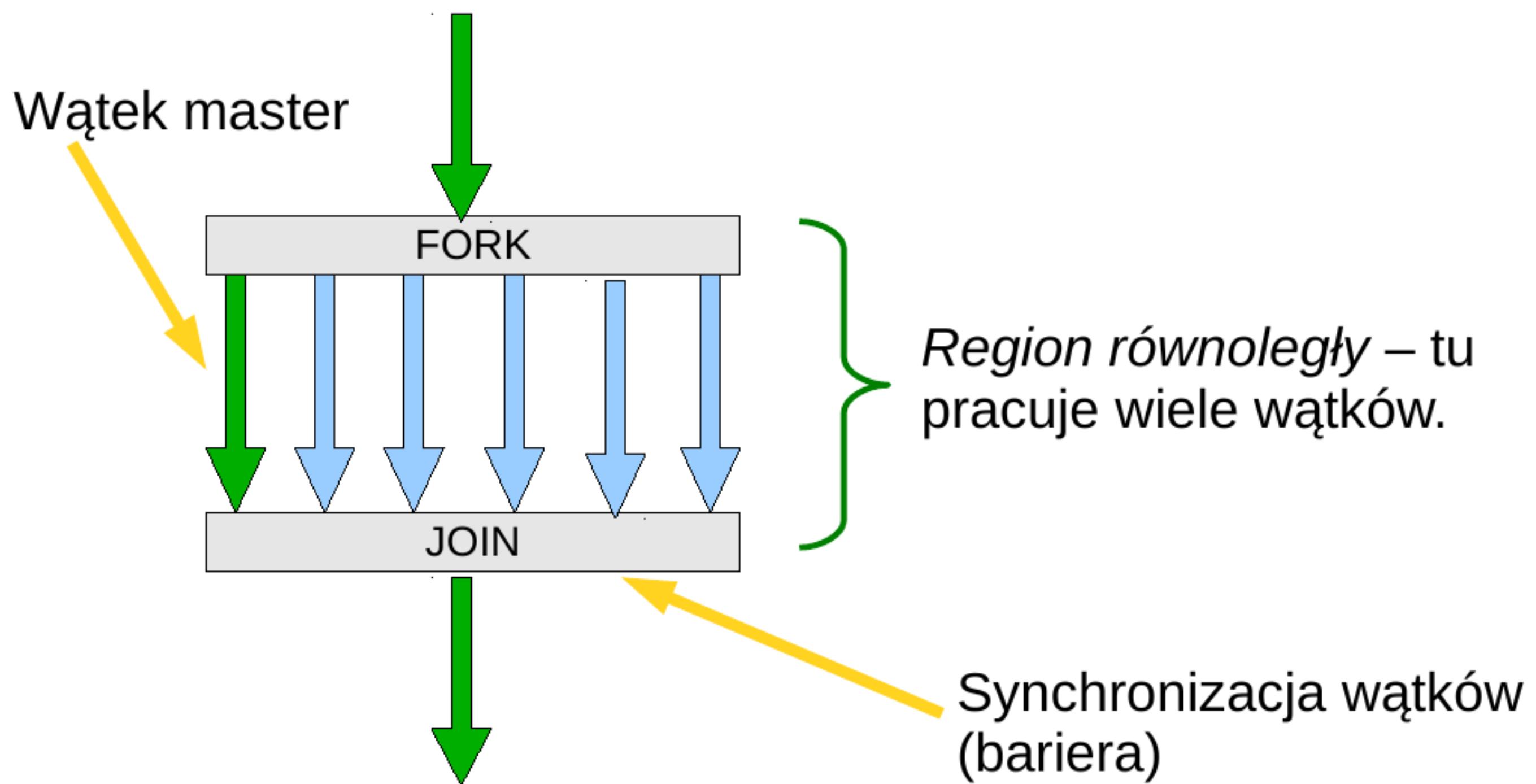
Wątki realizują zadania (tasks)

Obszar zadania (task region) – obszar zawierający cały kod napotkany w trakcie wykonywania zadania.

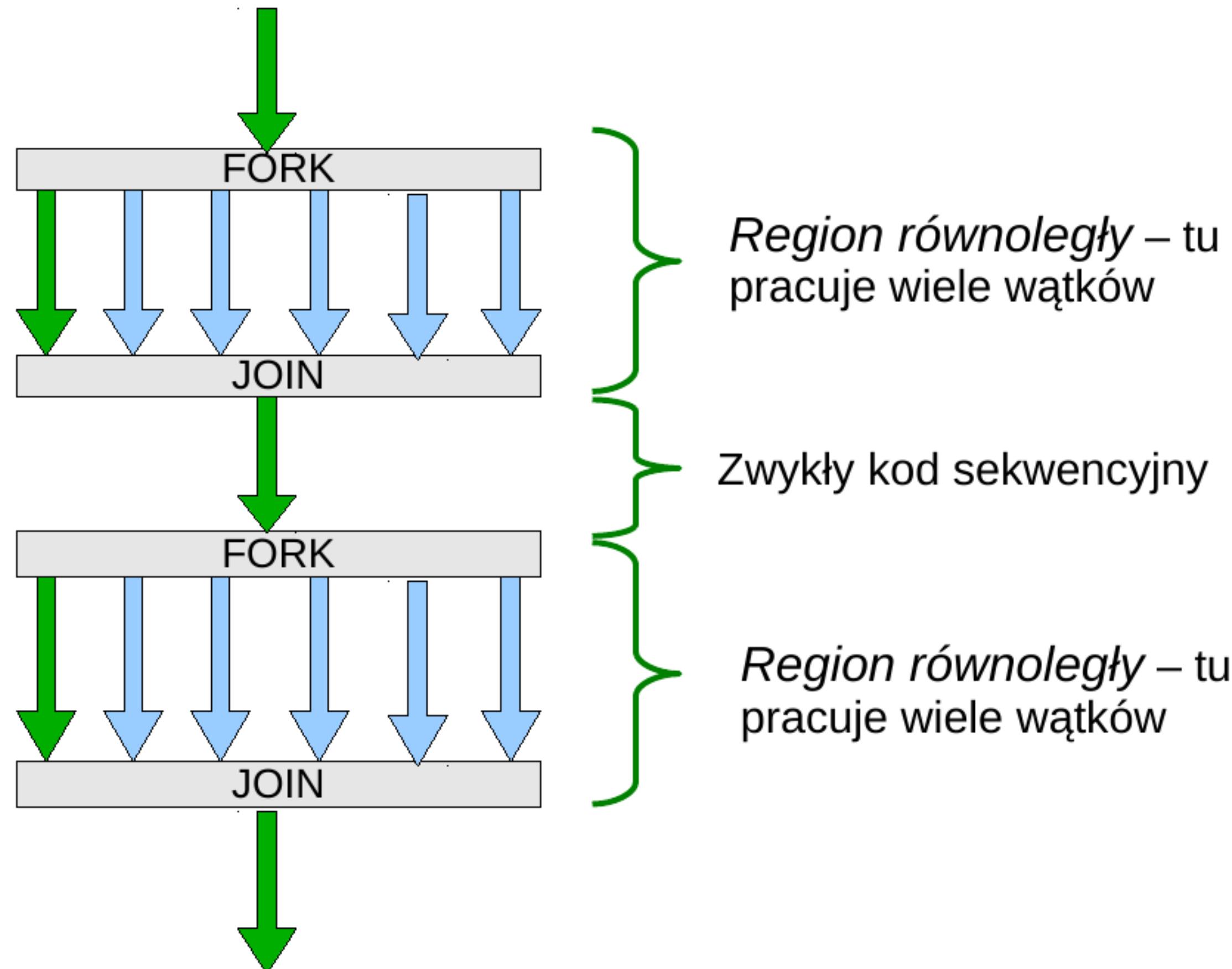
Zadanie związane (tied task) – jeśli zostanie wstrzymane w trakcie realizacji swojego obszaru zadania to może je podjąć tylko ten wątek, który to zadanie wstrzymał

Zadanie niezwiązane (untied task) – realizację wstrzymanego zadania może kontynuować każdy wątek z teamu realizującego dany obszar równoległy.

OpenMPI działa według modelu równoległości typu rozgałęzienie-połączenie (fork-join). Pojedynczy wątek rozgałęzia się w obszarach równoległych.



Choć tworzenie wątków kosztuje, robimy to wielokrotnie !



Ograniczenia równoległości na poziomie realizacji pojedynczych rozkazów - Instruction-Level Parallelism (ILS)

Od lat 80-tych XX wieku komputery wieloprocesorowe (kompletne procesory) i pamięć współdzielona.

Memory consistency problem!

Wyjątek: ccNUMA

Nawet w SMP pamięć cache nie jest współdzielona

Rdzenie w jednym CPU współdzielą pamięć cache

Model pamięci w OpenMP

- **relaxed-consistency shared-memory model**
- wątki mają dostęp do współdzielonej pamięci
- wątki mają prawo do tymczasowego widoku zmiennych
- widok tymczasowy nie jest wymagany
- widok tymczasowy pozwala na cache-owanie zmiennych
- wątek może posiadać prywatny obszar pamięci
- zmienne dzielą się na współdzielone i prywatne
- brak gwarancji, że operacje na zmiennych są atomowe
- tylko w pewnych dobrze-określonych punktach wątki mają zagwarantowane, że widzą te same wartości zmiennych współdzielonych.

Pomiędzy wątkiem a pamięcią mogą być rejestr i cache

To programista odpowiada za bezpieczeństwo danych

Synchronizacja pamięci

- widok tymczasowy pamięci oznacza czasowy brak konsystencji z pamięcią RAM
- flush – wymiatanie – wymuszenie konsystencji pamięci
- wykonanie operacji wymiecenia oznacza utratę widoku tymczasowego
- w trakcie wymiatania wszelkie operacje na wymiatanych zmiennych są zablokowane
- Aby dwa wątki mogły prawidłowo przekazać sobie dane przez współdzieloną zmienną w trakcie operacji wykonywanych na niej przez jeden wątek, drugi nie może jej używać do chwili wymiecenia.
- Zgodność widoku zmiennych trwa od chwili uzyskania konsystencji do pierwszej modyfikacji.

To programista odpowiada za bezpieczeństwo danych

Ogólny schemat dyrektyw OpenMP to:

```
#pragma omp konstrukcja [klauzula] [klauzula]
```

Dyrektwy OpenMP to dyrektywy preprocesora !

Najważniejsza dyrektywa OpenMP

```
#pragma omp parallel [klauzule...]
{
    // blok strukturalny C/C++
}
```

Blok strukturalny jest blokiem instrukcji C/C++ z pojedynczym wejściem i pojedynczym wyjściem.

Kod nie objęty dyrektywą **parallel** wykonywany jest sekwencyjnie.

Bez dodatkowych dyrektyw kod jest tylko replikowany i wszystkie wątki robią to sam...

Blok zakończony jest *implicite* barierą

Przykład:

```
#include <iostream>
int main ( void ) {
#pragma omp parallel
{
    std::cout << "Hello world!"
        << std::endl;
}
return 0;
}
```

```
> c++ -fopenmp code1.cpp
> env OMP_NUM_THREADS=4 ./a.out
Hello world!Hello world!Hello world!
```

Hello world!

Tworzymy aktywny region równoległy

Każdy wątek robi to samo!

Równoległe operacje input/output nie są synchroniczne

Jak to działa?

- konstrukcja **parallel** powoduje powstanie teamu wątków
- wątek, który napotkał **parallel** staje się masterem teamu wątków, w teamie jest on i zero lub więcej nowych wątków
- dla każdego wątku w teamie generowane jest niejawne zadanie
- wątki są wiązane z zadaniami (zadania związane)
- wykonywanie kodu z poprzedniego obszaru zadania jest wstrzymywane
- team wykonuje niejawne zadania
- wątki synchronizują się na niejawniej barierze
- poza blok objęty konstrukcją **parallel** wychodzi tylko wątek master
- wątek master powraca do wykonywania wstrzymanego zadania

Liczba wątków w teamie nie ulega zmianie w trakcie wykonywania bloku **parallel**

Prosty przykład użycia funkcji bibliotecznych:

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

using namespace std;

int main ( void ) {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    cout << "To ja watek nr " << id << " z "
        << nthreads << endl;
}
}
```

```
> env OMP_NUM_THREADS=4 ./a.out
```

```
To ja watek nr 3 z 4
```

```
To ja watek nr 2 z 4
```

```
To ja watek nr 1 z 4
```

```
To ja watek nr 0 z 4
```

Na maszynie
sekwencyjnej

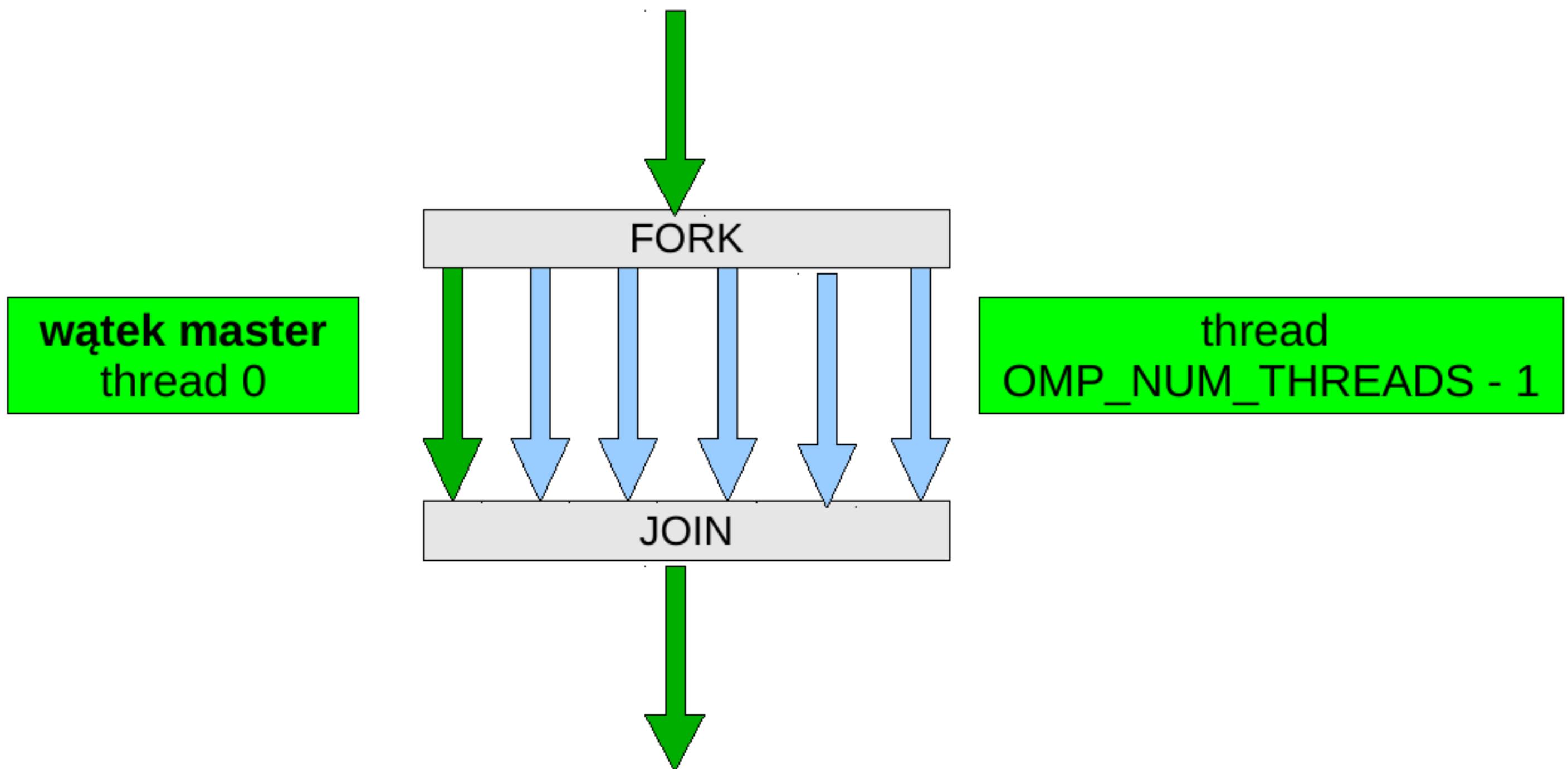
```
> env OMP_NUM_THREADS=4 ./a.out
```

```
To ja watek nr To ja watek nr 21 z 4 z 4
```

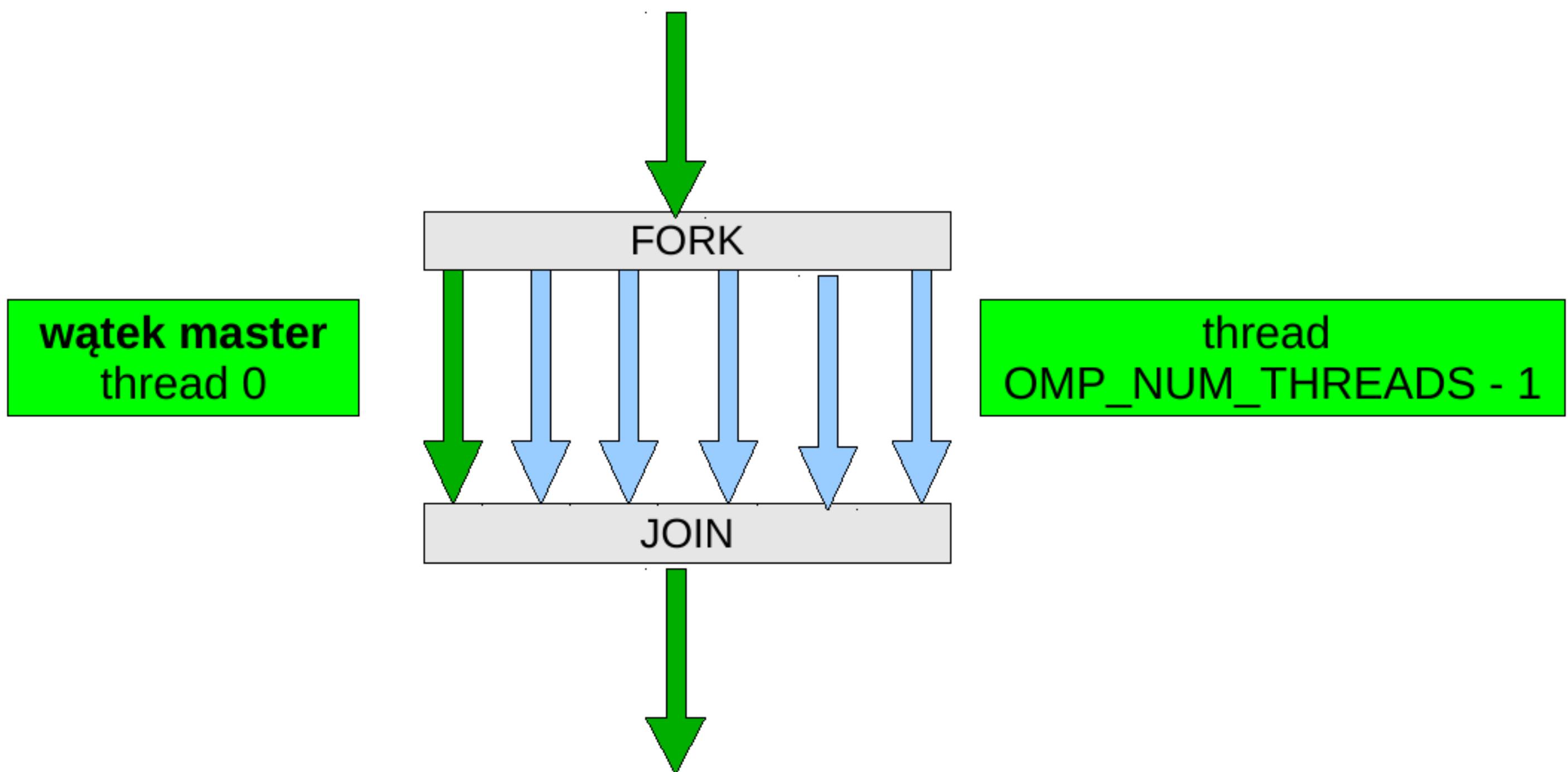
Na dual-core

```
To ja watek nr 3To ja watek nr 0 z z 44
```

Funkcja `omp_set_num_threads()` i zmienna systemowa `OMP_NUM_THREADS` decydują o ilości tworzonych wątków. Funkcja `omp_get_thread_num()` pozwala poznać numer wątku.



Użycie w programie funkcji **omp_get_thread_num()** pozwala przydzielić wątkom różne zadanie – taki kod może być najefektywniejszym rozwiązaniem, ale nie jest przenośny...



Użycie w programie *funkcji* biblioteki OpenMP spowoduje problemy, gdy kompilator nie zna OpenMP (mało prawdopodobne) lub wsparcie OpenMP nie zostało włączone.

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

using namespace std;

int main() {

    cout << "Liczba wątków " << omp_get_num_threads() << endl;

    return 0;
}
```

I komplikacja:

```
macro.cpp:(.text+0x20): undefined reference to `omp_get_num_threads'
collect2: error: ld returned 1 exit status
```

Błąd zgłasza linker

Rozwiązaniem jest użycie makra `_OPENMP`, które wskazuje na datę wydania wersji OpenMP, która jest przez kompilator wspierana.

```
#include<iostream>

using namespace std;

#ifndef _OPENMP
    #include<omp.h>
#else
    #define omp_get_num_threads() 1
    #define _OPENMP 0
#endif
```

```
int main() {
    cout << "Liczba wątków " << omp_get_num_threads() << endl;
    cout << "_OPENMP " << _OPENMP << endl;
    return 0;
}
```

Wynik:

```
c++ macro.cpp
Liczba wątków 1
_OPENMP 0
```

```
c++ -fopenmp macro.cpp
Liczba wątków 1
_OPENMP 201511
```

gcc wersja 7.3
Wsparcie dla
OpenMP 4.5

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

using namespace std;

int main ( void ) {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    cout << "To ja watek nr " << id << " z "
        << nthreads << endl;
    if ( id == 1 ) cout << "Ja mam id 1" << endl;
}
}
```

Na maszynie
sekwencyjnej

```
> env OMP_NUM_THREADS=4 ./a.out
To ja watek nr 3 z 4
To ja watek nr 1 z 4
Ja mam id 1
To ja watek nr 2 z 4
To ja watek nr 0 z 4
```

```
> env OMP_NUM_THREADS=4 ./a.out
To ja watek nr 3 z 4
To ja watek nr 2 z 4
To ja watek nr 1 z 4
Ja mam id 1
To ja watek nr 0 z 4
```

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

using namespace std;

int main ( void ) {
#pragma omp parallel
{
    id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    cout << "To ja watek nr " << id << " z "
        << nthreads << endl;
#pragma omp barrier
    if ( id == 1 ) cout << "Ja mam id 1" << endl;
}
}
```

```
> env OMP_NUM_THREADS=4 ./a.out
To ja watek nr 3 z 4
To ja watek nr 1 z 4
To ja watek nr 2 z 4
To ja watek nr 0 z 4
Ja mam id 1
```

Na maszynie
sekwencyjnej

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

int main ( void ) {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    std::cout << "A> To ja watek nr " << id << " z "
              << nthreads << std::endl;
#pragma omp barrier
    if ( id == 1 ) {
        std::cout << "Ja mam id 1" << std::endl;
//        return 0; error: invalid exit from OpenMP structured block
        int a = 1 / (id - 1);
    }
    std::cout << "B> To ja watek nr " << id << " z "
              << nthreads << std::endl;
}
}
```

Na maszynie
równoległej

```
> env OMP_NUM_THREADS=4 ./a.out
A> To ja watek nr A> To ja watek nr 12 z z 44
A> To ja watek nr A> To ja watek nr 03 z 4 z
4
Ja mam id 1
B> To ja watek nr 0 z 4
Błąd w obliczeniach zmiennoprzecinkowych
```

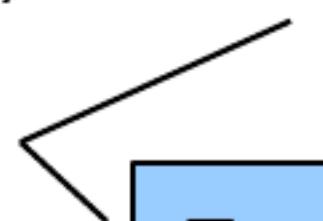
Jeden wątek
umiera to giną
wszystkie

Praca zlecona
przed barierą
wykona się
na pewno

**Ilość wykonanej
pracy zleconej
po barierze
jest nieokreślona**

Podział pracy:

- Praca do wykonywania jest dystrybuowana pomiędzy wątki
- Wykonywany jest w ramach niejawnych zadań
- Brak bariery na wejściu
- Niejawna bariera na wyjściu (można ją usunąć)
- Konstrukcje: obsługa pętli, **sections**, **single**, **workshare**



To dla języka
Fortran

Podstawowe warunki podziału pracy:

- Do obszaru podziału pracy wchodzą albo wszystkie wątki z teamu albo żaden.
- Obszary podziału pracy i bariery muszą być napotykane w tej samej kolejności

Dyrektywy rozdziału pracy muszą (dla uzyskania spodziewanego efektu) napotkać **aktywny obszar równoległy**.

A co z obszarami sekwencyjnymi i nieaktywnymi równoległymi ?

Aby było ciekawiej ten sam obszar oznaczony dyrektywami rozdziału pracy raz może być wykonany równolegle a chwilę później sekwencyjnie...

Dyrektywy rozdziału pracy nie tworzą nowych wątków

Chwila zastanowienia...

Czy poniższe pętle można zrównoleglić ze względu na iterację po zmiennej „i” ? Czy wątki mogą współdzielić zawartość pamięci przypisanej wszystkim zmiennym ?

```
for ( i = 1; i < 100; i++ )  
    for ( j = 0; j < i; j++ )  
        v[ i ][ j ] = v[ i - 1 ][ j ] + j / i;
```

Podział pracy pętli:

- praca przydzielana jest istniejącym w teamie wątkom w kontekście wykonywanych przez nie niewizualnych zadań
- wspierana jest tylko pętla for

Warunki:

- ilość iteracji musi być znana w chwili wejścia do bloku pętli
- ilość iteracji nie może podlegać zmianie w trakcie pracy tej pętli
- iteracje są niezależne (bez względu na kolejność wykonania wynik ma być ten sam)
- ograniczenie na typ zmiennej zliczającej iteracje

Zrównoleglenie pętli for()

```
#pragma omp for [klauzule...]
for( ; ; ) // pętla for
```

Iteracje są rozdzielane pomiędzy wątki.

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

using namespace std;

int main ( void ) {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int max = 6;

    for ( int i = 0; i < max; i++ ) {
        cout << "To ja watek nr " << id << " z "
            << nthreads << " wykonuje iteracje "
            << i << endl;
    } // for
} // omp parallel
} // main
```

```
> export OMP_NUM_THREADS=2  
> ./a.out
```

```
To ja watek nr 1 z 2 wykonuje iteracje 0  
To ja watek nr 1 z 2 wykonuje iteracje 1  
To ja watek nr 1 z 2 wykonuje iteracje 2  
To ja watek nr 1 z 2 wykonuje iteracje 3  
To ja watek nr 1 z 2 wykonuje iteracje 4  
To ja watek nr 1 z 2 wykonuje iteracje 5  
To ja watek nr 0 z 2 wykonuje iteracje 0  
To ja watek nr 0 z 2 wykonuje iteracje 1  
To ja watek nr 0 z 2 wykonuje iteracje 2  
To ja watek nr 0 z 2 wykonuje iteracje 3  
To ja watek nr 0 z 2 wykonuje iteracje 4  
To ja watek nr 0 z 2 wykonuje iteracje 5
```

Na maszynie
sekwencyjnej

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

using namespace std;

int main ( void ) {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int max = 6;
#pragma omp for
    for ( int i = 0; i < max; i++ ) {
        cout << "To ja watek nr " << id << " z "
            << nthreads << " wykonuje iteracje "
            << i << endl;
    } // for
} // omp parallel
} // main
```

```
> export OMP_NUM_THREADS=2  
> ./a.out
```

```
To ja watek nr 0 z 2 wykonuje iteracje 0  
To ja watek nr 0 z 2 wykonuje iteracje 1  
To ja watek nr 0 z 2 wykonuje iteracje 2  
To ja watek nr 1 z 2 wykonuje iteracje 3  
To ja watek nr 1 z 2 wykonuje iteracje 4  
To ja watek nr 1 z 2 wykonuje iteracje 5
```

Na maszynie
sekwencyjnej

```
> export OMP_NUM_THREADS=11  
> ./a.out
```

?

```
> export OMP_NUM_THREADS=11
```

```
> ./a.out
```

To ja watek nr 3 z 11 wykonuje iteracje 3

To ja watek nr 4 z 11 wykonuje iteracje 4

To ja watek nr 5 z 11 wykonuje iteracje 5

To ja watek nr 2 z 11 wykonuje iteracje 2

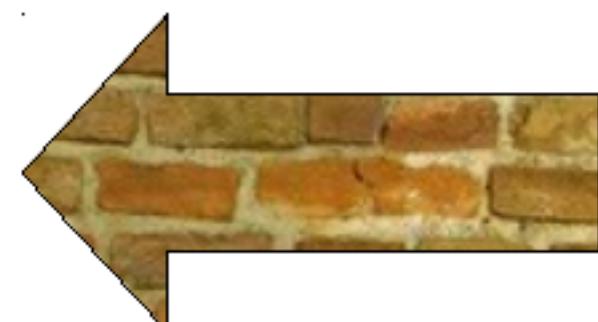
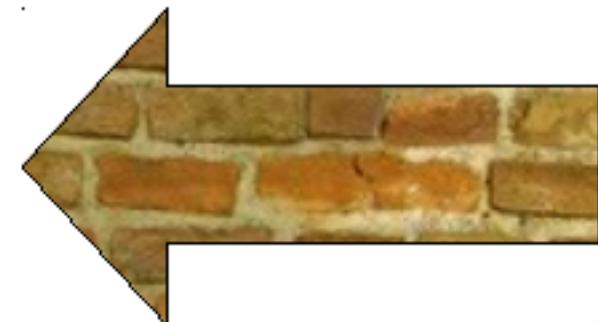
To ja watek nr 1 z 11 wykonuje iteracje 1

To ja watek nr 0 z 11 wykonuje iteracje 0

Na maszynie
sekwencyjnej

```
int main ( void ) {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int max = 6;
#pragma omp for
    for ( int i = 0; i < max; i++ ) {
        cout << "To ja watek nr " << id << " z "
            << nthreads << " wykonuje iteracje "
            << i << endl;
    } // for
#pragma omp for
    for ( int i = 0; i < max; i++ ) {
        cout << "To ja watek nr " << id << " z "
            << nthreads << " wykonuje iteracje2 " << i
            << endl;
    } // for
} // omp parallel
} // main
```

```
> export OMP_NUM_THREADS=4
> ./a.out
To ja watek nr 1 z 4 wykonuje iteracje 2
To ja watek nr 1 z 4 wykonuje iteracje 3
To ja watek nr 2 z 4 wykonuje iteracje 4
To ja watek nr 2 z 4 wykonuje iteracje 5
To ja watek nr 0 z 4 wykonuje iteracje 0
To ja watek nr 0 z 4 wykonuje iteracje 1
To ja watek nr 2 z 4 wykonuje iteracje2 4
To ja watek nr 2 z 4 wykonuje iteracje2 5
To ja watek nr 1 z 4 wykonuje iteracje2 2
To ja watek nr 1 z 4 wykonuje iteracje2 3
To ja watek nr 0 z 4 wykonuje iteracje2 0
To ja watek nr 0 z 4 wykonuje iteracje2 1
```



Zrównoleglenie poprzez przydział wątkom różnych bloków kodu

```
#pragma omp sections [klauzule...]
{
    [#pragma omp section]
    {
        Blok strukturalny C/C++
    }
    [#pragma omp section]
    {
        Blok strukturalny C/C++
    }
}
```

- Uwaga na niebalansowanie obciążenia !!!!
- Nie ma gwarancji w jakiej kolejności wykonane zostaną bloki – nawet w przypadku jednego wątku !!!
- Na końcu jest niejawną barierą, którą można usunąć.
- Konstrukcja section tylko w sections

```
int main ( void ) {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int max = 6;
#pragma omp sections
    {
#pragma omp section
        for ( int i = 0; i < max; i++ ) {
            cout << "To ja watek nr " << id << " z "
                << nthreads << " wykonuje iteracje "
                << i << endl;
        }
#pragma omp section
        for ( int i = 0; i < max; i++ ) {
            cout << "To ja watek nr " << id << " z "
                << nthreads << " wykonuje iteracje2 "
                << i << endl;
        }
    }
}
```

```
> export OMP_NUM_THREADS=4
```

```
> ./a.out
```

```
To ja watek nr 3 z 4 wykonuje iteracje 0
To ja watek nr 1 z 4 wykonuje iteracje2 0
To ja watek nr 1 z 4 wykonuje iteracje2 1
To ja watek nr 1 z 4 wykonuje iteracje2 2
To ja watek nr 1 z 4 wykonuje iteracje2 3
To ja watek nr 1 z 4 wykonuje iteracje2 4
To ja watek nr 1 z 4 wykonuje iteracje2 5
To ja watek nr 3 z 4 wykonuje iteracje 1
To ja watek nr 3 z 4 wykonuje iteracje 2
To ja watek nr 3 z 4 wykonuje iteracje 3
To ja watek nr 3 z 4 wykonuje iteracje 4
To ja watek nr 3 z 4 wykonuje iteracje 5
```

Na maszynie
sekwencyjnej

Fragment kodu realizowany wyłącznie przez jeden wątek

```
#pragma omp single [klauzule...]
{
    Blok strukturalny C/C++
}
```

Jeden wątek pracuje reszta czeka...

Na końcu bloku jest bariera, ale można ją usunąć

Do czego tego dziwadła używać ?

Aby oszczędzać klawiaturę można stosować skróty

```
#pragma omp parallel for [klauzule...]
```

```
#pragma omp parallel sections [klauzule...]
```

A jak z efektywnością kodu ?

Czas by poznać klauzule – dodatki do dyrektyw, które instruują kompilator o sposobie radzenia sobie z danym blokiem kodu

```
#pragma omp konstrukcja [klauzule...]
{
    Blok kodu zawierający zmienne...
}
```

Klauzula **collapse**(liczba pętli)

Pozwala połączyć iteracje pętli zagnieżdżonych i rozdzielić pracę z większej przestrzeni iteracji

```
#pragma omp parallel
```

```
{
```

```
    int id = omp_get_thread_num();  
    int max = 4;
```

```
#pragma omp for collapse(2)
```

```
    for ( int i = 0; i < max; i++ )  
        for ( int j = 0; j < max; j++ )  
            cout << "To ja watek nr " << id  
                << " wykonuje iteracje ( " << i << ", " << j  
                << ")" << endl;  
}
```

Bez collapse

Na maszynie
sekwencyjnej

```
> env OMP_NUM_THREADS=3 ./a.out
```

```
To ja watek nr 0 wykonuje iteracje ( 0, 0)
To ja watek nr 0 wykonuje iteracje ( 0, 1)
To ja watek nr 0 wykonuje iteracje ( 0, 2)
To ja watek nr 0 wykonuje iteracje ( 0, 3)
To ja watek nr 0 wykonuje iteracje ( 1, 0)
To ja watek nr 0 wykonuje iteracje ( 1, 1)
To ja watek nr 0 wykonuje iteracje ( 1, 2)
To ja watek nr 0 wykonuje iteracje ( 1, 3)
To ja watek nr 1 wykonuje iteracje ( 2, 0)
To ja watek nr 1 wykonuje iteracje ( 2, 1)
To ja watek nr 1 wykonuje iteracje ( 2, 2)
To ja watek nr 1 wykonuje iteracje ( 2, 3)
To ja watek nr 1 wykonuje iteracje ( 3, 0)
To ja watek nr 1 wykonuje iteracje ( 3, 1)
To ja watek nr 1 wykonuje iteracje ( 3, 2)
To ja watek nr 1 wykonuje iteracje ( 3, 3)
```

Z collapse

Na maszynie
sekwencyjnej

```
> env OMP_NUM_THREADS=3 ./a.out
```

```
To ja watek nr 0 wykonuje iteracje ( 0, 0)
To ja watek nr 0 wykonuje iteracje ( 0, 1)
To ja watek nr 0 wykonuje iteracje ( 0, 2)
To ja watek nr 0 wykonuje iteracje ( 0, 3)
To ja watek nr 0 wykonuje iteracje ( 1, 0)
To ja watek nr 0 wykonuje iteracje ( 1, 1)
To ja watek nr 2 wykonuje iteracje ( 3, 0)
To ja watek nr 2 wykonuje iteracje ( 3, 1)
To ja watek nr 2 wykonuje iteracje ( 3, 2)
To ja watek nr 2 wykonuje iteracje ( 3, 3)
To ja watek nr 1 wykonuje iteracje ( 1, 2)
To ja watek nr 1 wykonuje iteracje ( 1, 3)
To ja watek nr 1 wykonuje iteracje ( 2, 0)
To ja watek nr 1 wykonuje iteracje ( 2, 1)
To ja watek nr 1 wykonuje iteracje ( 2, 2)
To ja watek nr 1 wykonuje iteracje ( 2, 3)
```

Klauzula **shared**(lista zmiennych)

Wskazuje współdzielone pomiędzy wątkami zmienne.

Współdzielona zmienna istnieje w jednej instancji a wątki mają prawo do jej odczytu i modyfikacji (nawet jednocześnie)

```
#pragma omp parallel for shared( a )  
for ( i = 0; i < 100; i++ ) {  
    a[ i ] += i;
```

Klauzula **private**(lista zmiennych)

Wskazuje zmienne prywatne wątku

Każdy wątek dysponuje własną kopią zmiennych z listy i ma wyłączny dostęp do własnych kopii. Zmiany dokonane przez jeden wątek nie są widoczne dla innych.

Zmienne zadeklarowane wewnątrz bloku objętego dyrektywą OpenMP są domyślnie prywatne

```
#pragma omp parallel for private( i )
for ( i = 0; i < 100; i++ ) {
    a[ i ] += i;
```

Klauzula **private**(lista zmiennych)

Wskazuje zmienne prywatne wątku

Uwaga: zmienne prywatne są niezdefiniowane przed wejściem do bloku i po jego opuszczeniu.

Zmienne zdefiniowane przed wejściem do bloku a znajdującej się na liście private po wyjściu będą nieokreślone.

Uwaga: zmienne zadeklarowane w bloku parallel automatycznie stają się prywatnymi. Tej klauzuli należy używać, gdy zmienna zadeklarowana została przed blokiem parallel i domyślnie zostanie uznana za współdzieloną.

Klauzula **lastprivate**(lista zmiennych)

Wskazuje zmienne prywatne wątku, które po zakończeniu bloku oznaczonego jako `#pragma omp for` lub `#pragma omp sections` przyjmują wartość taką jak w przypadku sekwencyjnego wykonania kodu. Najpierw zwykłe **private**.

```
int i = 123;  
#pragma omp parallel for private(i)  
for ( i = 0; i < 100; i++ ) {  
    cout << "Iteracja i " << i  
    << " wykonana przez : " << omp_get_thread_num()  
    << endl;  
}  
cout << "i = " << i << endl;  
[...]  
Iteracja i 49 wykonana przez : 1  
i = 123
```

Od wersji > 2.5
Wcześniej wartość
nieokreślona!

Klauzula **lastprivate**(lista zmiennych)

Wskazuje zmienne prywatne wątku, które po zakończeniu bloku oznaczonego jako `#pragma omp for` lub `#pragma omp sections` przyjmują wartość taką jak w przypadku sekwencyjnego wykonania kodu

```
int i = 123;  
#pragma omp parallel for lastprivate(i)  
for ( i = 0; i < 100; i++ ) {  
    cout << "Iteracja i " << i  
        << " wykonana przez : " << omp_get_thread_num()  
        << endl;  
}  
cout << "i = " << i << endl;  
  
[...]  
Iteracja i 49 wykonana przez : 1  
i = 100
```

Klauzula **firstprivate**(lista zmiennych)

Wskazuje zmienne prywatne wątku, które są inicjowane zmiennymi o identycznej nazwie zadeklarowanymi przed blokiem

```
int i = 5;  
#pragma omp parallel firstprivate(i)  
{  
    i += omp_get_thread_num();  
    cout << "Hej moja wartosc i to " << i  
        << " a moje id : " << omp_get_thread_num()  
        << endl;  
}
```

```
Hej moja wartosc i to 8 a moje id : 3  
Hej moja wartosc i to 6 a moje id : 1  
Hej moja wartosc i to 7 a moje id : 2  
Hej moja wartosc i to 5 a moje id : 0
```

Klauzula **default**(shared | none)

Wskazuje domyślny atrybut dla zmiennych (shared) lub wymusza (none) podanie atrybutów dla wszystkich zmiennych

```
#pragma omp parallel for default(shared) private(i)
for ( i = 0; i < 100; i++ )
{
    B[ i ] = i / 100.0;
    C[ i ] = i + 1;
    A[ i ] = B[ i ] + C[ i ];
}
```

Klauzula **nowait**

Usuwa barierę na zakończenie bloku rozdzielającego pracę.

Bariera na końcu bloku *#pragma omp parallel* jest nieusuwalna.

#pragma omp for nowait

```
for ( int i = 0; i < max; i++ ) {
    cout << "To ja watek nr " << id << " z "
        << nthreads << " wykonuje iteracje "
        << i << endl;
} // for
#pragma omp for
for ( int i = 0; i < max; i++ ) {
    cout << "To ja watek nr " << id << " z "
        << nthreads << " wykonuje iteracje2 " << i
        << endl;
} // for
```

```
> export OMP_NUM_THREADS=4
> ./a.out
To ja watek nr 1 z 4 wykonuje iteracje 2
To ja watek nr 1 z 4 wykonuje iteracje 3
To ja watek nr 1 z 4 wykonuje iteracje2 2
To ja watek nr 1 z 4 wykonuje iteracje2 3
To ja watek nr 2 z 4 wykonuje iteracje 4
To ja watek nr 2 z 4 wykonuje iteracje 5
To ja watek nr 2 z 4 wykonuje iteracje2 4
To ja watek nr 2 z 4 wykonuje iteracje2 5
To ja watek nr 0 z 4 wykonuje iteracje 0
To ja watek nr 0 z 4 wykonuje iteracje 1
To ja watek nr 0 z 4 wykonuje iteracje2 0
To ja watek nr 0 z 4 wykonuje iteracje2 1
```

Klauzula **schedule(rodzaj [, porcja])**

Określa jak iteracje pętli są przypisywane wątkom, czyli sposób podziału większej pracy na mniejsze.

schedule(static [,]) iteracje są przypisywane wątkom statycznie stałymi porcjami wg. algorytmu karuzelowego. Gdy rozmiar porcji iteracji nie jest podany iteracje dzielone są w miarę możliwości po równo.

schedule(dynamic [,]) iteracje są przypisywane wątkom w porcjach gdy wątek o nie poprosi. Domyślna porcja to jedna iteracja

schedule(guided [,]) iteracje są przypisywane wątkom w porcjach gdy wątek o nie poprosi. Ilość iteracji przypisywana dla wątku to ilość iteracji do wykonania podzielona przez liczbę wątków. Przypisywana ilość iteracji zmniejsza się do podanego rozmiaru porcji (domyślnie 1).

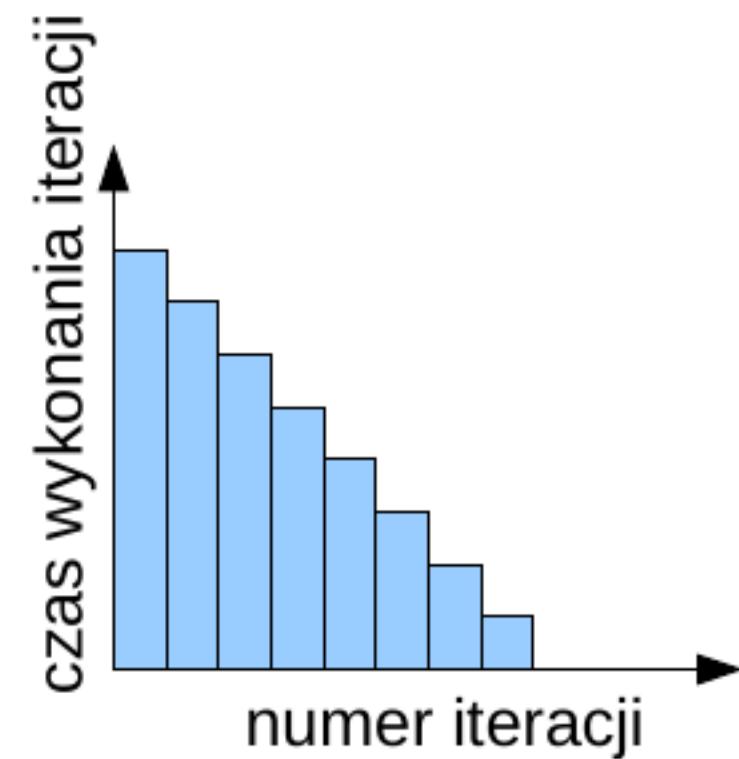
Klauzula **schedule**(rodzaj [, porcja])

schedule(runtime) sposób zarządzania iteracjami jest do odczytania ze zmiennej środowiskowej OMP_SCHEDULE

Klauzula **schedule(rodzaj [, porcja])**

Przykład problemu: ilość pracy do wykonania spada wraz ze wzrostem zmiennej „i”.

```
#pragma omp for
for ( i = 0; i < 100; i ++ )
    for ( j = i; j < 100; j++ )
{
    // coś super ważnego
}
```



Wraz z „i” ilość czasu potrzebnego na wykonanie jednej iteracji może również maleć, zmieniać się nagle, zachowywać niemal losowo itd. Na ćwiczeniach sprawdzimy jak sobie w takich sytuacjach radzić.

KONIEC