SZKOLENIE ICM

Programowanie równoległe: OpenMP

Michał Hermanowicz
m.hermanowicz@icm.edu.pl

Interdyscyplinarne Centrum Modelowania Matematycznego i Komputerowego
Uniwersytet Warszawski

10 listopada 2022

1. Wprowadzenie do OpenMP
   - Obliczenia równoległe i pamięć współdzielona
   - Komponenty OpenMP
   - Model *fork-join*
   - Struktury kontrolne
   - Środowisko danych
   - Synchronizacja
   - Pętle `for` (`do`) i klauzula `COLLAPSE`
   - Dyrektywy SECTIONS, MASTER, SINGLE
   - Zadania (*tasks*)

2. Sesja praktyczna

Wprowadzenie

Literatura:

- The OpenMP Architecture Review Board (`https://www.openmp.org`)
  - Specifications (OpenMP API 5.1)
  - Reference Guides
  - Tutorials & Articles
- Rohit Chandra et al. *Parallel Programming in OpenMP*, Morgan Kaufmann, 2000
- Barbara Chapman et al. *Using OpenMP*, The MIT Press, 2008

Wprowadzenie

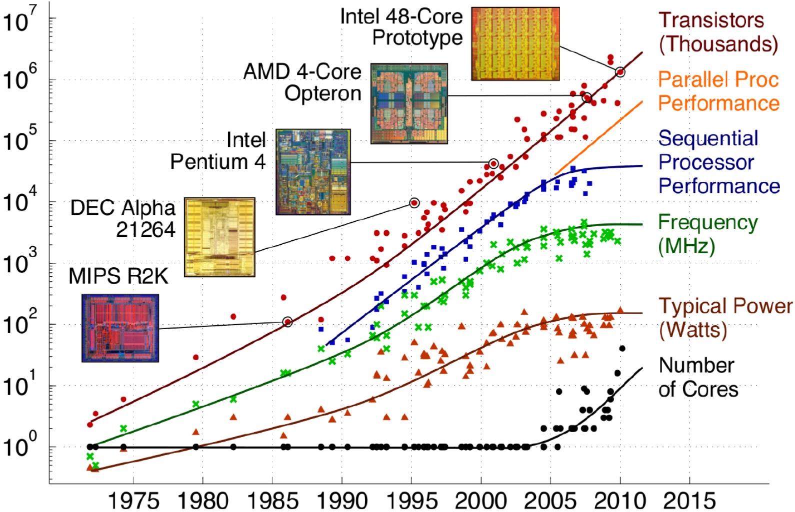
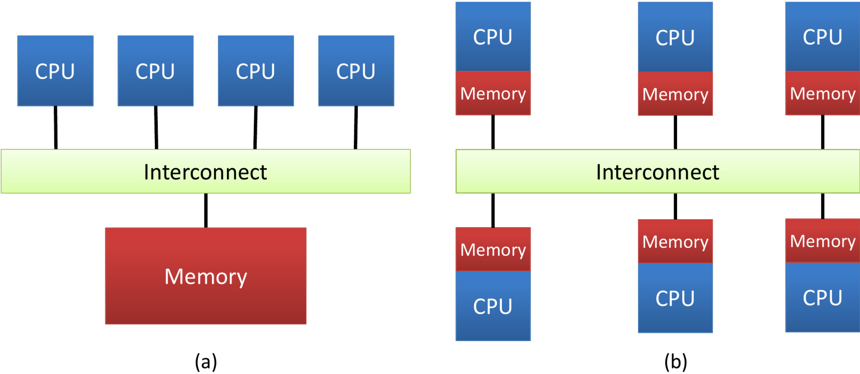


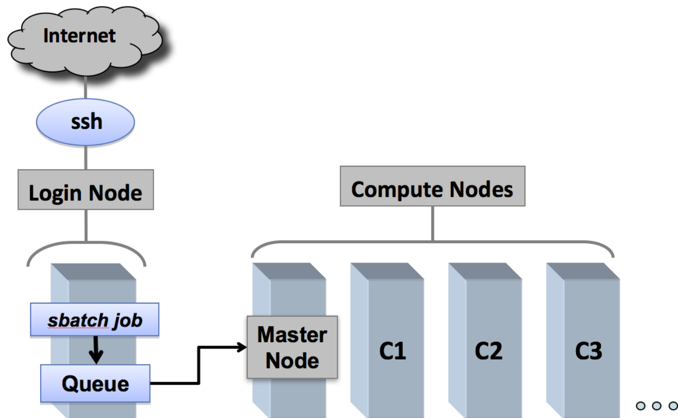
Figure: Christopher Batten, ECE 5950 Complex Digital ASIC Design Course Overview

Wprowadzenie: Pamięć współdzielona i rozproszona



Źródło: **R. Busseuil** et al. `DOI: 10.1007/978-3-642-28566-0_10`

Wprowadzenie: Infrastruktura HPC



Cornell Virtual Workshop (https://cvw.cac.cornell.edu/environment/slurm_intro)

Wprowadzenie

Czym jest OpenMP?

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs.

OpenMP is suitable for a wide range of algorithms running on multicore nodes and chips, NUMA systems, GPUs, and other such devices attached to a CPU.

Źródła: OpenMP API FAQ; Syntax Reference Guide (`https://www.openmp.org`)

Wprowadzenie

Czym jest OpenMP?

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs.

OpenMP is suitable for a wide range of algorithms running on multicore nodes and chips, NUMA systems, GPUs, and other such devices attached to a CPU.

Źródła: OpenMP API FAQ; Syntax Reference Guide (https://www.openmp.org)

W praktyce:

Zbiór dyrektyw i rutyn opisany przez API → **OpenMP**

Wprowadzenie

OpenMP:

- *nie jest* językiem programowania,
- współpracuje ze standardowymi językami Fortran, C/C++,
- opiera się na dyrektywach kompilatora, który opisuje mechanizm zrównoleglania na poziomie kodu źródłowego,
- dostarcza biblioteki uruchomieniowej (subrutyn dostępnych dla aplikacji).

Wprowadzenie: Charakterystyka OpenMP

- Programy wykorzystują *wątki*, każdy z nich ma dostęp do wspólnej pamięci,

Wprowadzenie: Charakterystyka OpenMP

- Programy wykorzystują *wątki*, każdy z nich ma dostęp do wspólnej pamięci,
- Zmienne mogą mieć atrybut `private` – wówczas dostępne dla danego wątku,

Wprowadzenie: Charakterystyka OpenMP

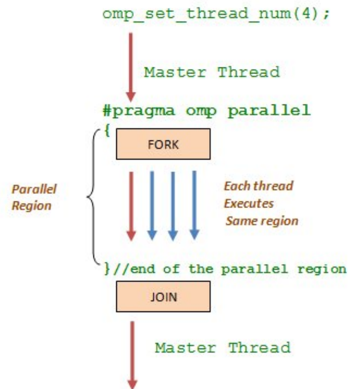
- Programy wykorzystują *wątki*, każdy z nich ma dostęp do wspólnej pamięci,
- Zmienne mogą mieć atrybut `private` – wówczas dostępne dla danego wątku,
- Wymiana danych odbywa się za pomoca operacji na zmiennych (współdzielonych),

Wprowadzenie: Charakterystyka OpenMP

- Programy wykorzystują *wątki*, każdy z nich ma dostęp do wspólnej pamięci,
- Zmienne mogą mieć atrybut `private` – wówczas dostępne dla danego wątku,
- Wymiana danych odbywa się za pomoca operacji na zmiennych (współdzielonych),
- Program działa w modelu `fork-join`.

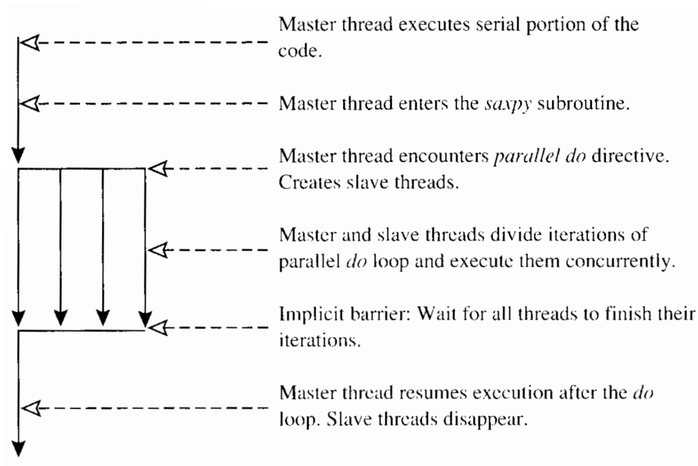
Wprowadzenie: Charakterystyka OpenMP

- Programy wykorzystują *wątki*, każdy z nich ma dostęp do wspólnej pamięci,
- Zmienne mogą mieć atrybut `private` – wówczas dostępne dla danego wątku,
- Wymiana danych odbywa się za pomoca operacji na zmiennych (współdzielonych),
- Program działa w modelu `fork-join`.



Model `fork-join`. Copyright (C) A.A. Aydin

Wprowadzenie: Model fork-join



R. Chandra et al. *Parallel Programming in OpenMP*

Wprowadzenie: Podstawowe komponenty OpenMP

Komponenty OpenMP:

- Dyrektywy,
- Subrutyny (biblioteka / *runtime library*),
- Zmienne środowiskowe.

Wprowadzenie: Podstawowe komponenty OpenMP

Komponenty OpenMP:

- Dyrektywy,
- Subrutyny (biblioteka / *runtime library*),
- Zmienne środowiskowe.

Dyrektywy: Fortran

```
!$omp ...       (fixed/free-form)
c$omp ...       (fixed-form)
*$omp ...       (fixed-form)
```

Dyrektywy: C/C++

```
#pragma omp ...
```

Wprowadzenie: Podstawowe komponenty OpenMP

Komponenty OpenMP:

- Dyrektywy,
- Subrutyny (biblioteka / *runtime library*),
- Zmienne środowiskowe.

Dyrektywy: Fortran

```
!$omp ...        (fixed/free-form)
c$omp ...        (fixed-form)
*$omp ...        (fixed-form)
```

Dyrektywy: C/C++

```
#pragma omp ...
```

Dyrektywy:

- mogą być kontynuowane w kolejnym wierszu (konieczny znak & na końcu pierwszej linii;
- w kodzie *fixed-form* (Fortran) muszą zaczynać się w pierwszej kolumnie;
- są ignorowane przez kompilatory, które nie wspierają OpenMP.

Wprowadzenie

Kompilacja warunkowa powoduje, że wybrane instrukcje są kompilowane wyłącznie w sytuacji, gdy OpenMP jest włączone – wymaga użycia prefiksu `!$` w przypadku Fortranu (zasady jak przy dyrektywach) i makra `_OPENMP` w przypadku C/C++.

Wprowadzenie

Kompilacja warunkowa powoduje, że wybrane instrukcje są kompilowane wyłącznie w sytuacji, gdy OpenMP jest włączone – wymaga użycia prefiksu `!$` w przypadku Fortranu (zasady jak przy dyrektywach) i makra `_OPENMP` w przypadku C/C++.

Wykonanie programu OpenMP – najważniejsze zagadnienia:

- model *fork-join*,
- struktury kontrolne (`parallel`, `do`),
- wątek główny (*master thread*),
- wątki równoległe (*fork*),
- zmienne prywatne/współdzielone,
- komunikacja pomiędzy wątkami,
- synchronizacja (`critical`, `barrier`).

Wprowadzenie

```fortran
program hello

  use omp_lib
  implicit none

  print *, "Hello World from threads:"

  !$omp parallel
  print *, omp_get_thread_num()
  !$omp end parallel

  print *, "Back to sequential run"

end program hello
```

Wprowadzenie

```
1  program hello
2
3    use omp_lib
4    implicit none
5
6    print *, "Hello World from threads:"
7
8    !$omp parallel
9    print *, omp_get_thread_num()
10   !$omp end parallel
11
12   print *, "Back to sequential run"
13
14 end program hello
```

```
(--ntasks-per-node=1 --cpus-per-task=4)
$ export OMP_NUM_THREADS = 4
$ gfortran -fopenmp hello.f90 -o hello
$ ./a.out
```

Wprowadzenie

```fortran
program hello

  use omp_lib
  implicit none

  print *, "Hello World from threads:"

  !$omp parallel
  print *, omp_get_thread_num()
  !$omp end parallel

  print *, "Back to sequential run"

end program hello
```

```
(--ntasks-per-node=1 --cpus-per-task=4)
$ export OMP_NUM_THREADS = 4
$ gfortran -fopenmp hello.f90 -o hello
$ ./a.out
```

```
Hello World from threads:
0
3
1
2
Back to sequential run
```

Wprowadzenie

```fortran
program hello

  use omp_lib
  implicit none

  print *, "Hello World from threads:"

  !$omp parallel
  print *, omp_get_thread_num()
  !$omp end parallel

  print *, "Back to sequential run"

end program hello
```

```
(--ntasks-per-node=1 --cpus-per-task=4)
$ export OMP_NUM_THREADS = 4
$ gfortran -fopenmp hello.f90 -o hello
$ ./a.out
```

```
Hello World from threads:
0
3
1
2
Back to sequential run
```

1. Master thread (sequential)
2. Fork (parallel)
3. Join to master thread (sequential)

Wprowadzenie

- Ile procesów powstaje w trakcie *forkowania*?

Wprowadzenie

- Ile procesów powstaje w trakcie *forkowania*?
  → Zmienne środowiskowe (`OMP_NUM_THREADS`)

Wprowadzenie

- Ile procesów powstaje w trakcie *forkowania*?
  → Zmienne środowiskowe (`OMP_NUM_THREADS`)
- W językach C/C++ → pragmy, plik nagłówkowy `omp.h`

Wprowadzenie

- Ile procesów powstaje w trakcie *forkowania*?
  → Zmienne środowiskowe (`OMP_NUM_THREADS`)
- W językach C/C++ → pragmy, plik nagłówkowy `omp.h`
- Program OpenMP można zbudować kompilatorem, który nie wspiera OpenMP: Dyrektywy z nim związane będą wówczas ignorowane. Ale: dotyczy to tylko dyrektyw; może zachodzić potrzeba kompilacji warunkowej.

Wprowadzenie

- Ile procesów powstaje w trakcie *forkowania*?
  → Zmienne środowiskowe (`OMP_NUM_THREADS`)
- W językach C/C++ → pragmy, plik nagłówkowy `omp.h`
- Program OpenMP można zbudować kompilatorem, który nie wspiera OpenMP: Dyrektywy z nim związane będą wówczas ignorowane. Ale: dotyczy to tylko dyrektyw; może zachodzić potrzeba kompilacji warunkowej.

Kompilacja warunkowa

Jeżeli program zawiera wyrażenia specyficzne dla wersji równoległej, to nie powinny (nie mogą) one być interpretowane w wersji sekwencyjnej.

Potrzebny prefix: `!$`, `c$`, `*$` (fixed-form Fortran)
lub
`!$` (free-form Fortran). Zasady składni jak przy innych dyrektywach.

Kompilacja warunkowa

```
1    (...)
2
3    implicit none
4    use lib_omp
5
6    !$ i = omp_get_thread_num()
7
8    y = x !$ + b
9
10   (...)
```

Kompilacja warunkowa

```fortran
(...)

implicit none
use lib_omp

!$ i = omp_get_thread_num()

y = x !$ + b

(...)
```

```fortran
(...)

implicit none
use lib_omp

!$ i = omp_get_thread_num()

y = x &
!$ + b

(...)
```

Kompilacja warunkowa

```
1    (...)
2
3    implicit none
4    use lib_omp
5
6    !$ i = omp_get_thread_num()
7
8    y = x !$ + b
9
10   (...)
```

```
1    (...)
2
3    implicit none
4    use lib_omp
5
6    !$ i = omp_get_thread_num()
7
8    y = x &
9    !$ + b
10
11   (...)
```

W języku C/C++ mamy do dyspozycji makro `_OPENMP`.

Struktury kontrolne

Trzy kategorie elementów OpenMP:

- Struktury kontrolne (*parallel control structures*),
- Środowisko danych (*data environment*),
- Synchronizacja (*synchronisation*).

Struktury kontrolne

kontrolują przebieg programu: tworzą nowe wątki w modelu fork/join, etc. Dyrektywy: `parallel` (tworzy wątki, które wykonują blok kodu równolegle) i `do` (*loop-level parallelism*) (dzieli zadania między wątki).

(Implicit barrier!)

Środowisko danych

Środowisko danych (*execution context*)

Wątek główny (*master thread*) jest inicjowany w pewnym kontekście danych: zmienne globalne, lokalne (subrutyny), alokowane dynamicznie. Ten kontekst jest aktywny przez cały czas działania programu (jego wątku głównego).

Środowisko danych

Środowisko danych (*execution context*)

Wątek główny (*master thread*) jest inicjowany w pewnym kontekście danych: zmienne globalne, lokalne (subrutyny), alokowane dynamicznie. Ten kontekst jest aktywny przez cały czas działania programu (jego wątku głównego).

Każdy utworzony wątek posiada własny stos, który jest prywatny dla każdej swojej subrutyny (zmienne nie są nadpisywane przez równolegle wykonywane wątki).

Środowisko danych

Środowisko danych (*execution context*)

Wątek główny (*master thread*) jest inicjowany w pewnym kontekście danych: zmienne globalne, lokalne (subrutyny), alokowane dynamicznie. Ten kontekst jest aktywny przez cały czas działania programu (jego wątku głównego).

Każdy utworzony wątek posiada własny stos, który jest prywatny dla każdej swojej subrutyny (zmienne nie są nadpisywane przez równolegle wykonywane wątki).

Zasięg zmiennej może być:

- `shared` – jeden adres w pamięci wspólny dla wszystkich wątków. Łatwa komunikacja pomiędzy wątkami przez proste operacje na zmiennej;
- `private` – wiele adresów w pamięci, po jednym dla kazdego wątku;
- `reduction` – ma cechy obu powyższych.

Synchronizacja

Synchronizacja

Komunikacja między wątkami odbywa się przez operacje na zmiennych współdzielonych.
Synchronizacja pozwala uniknąć konfliktów w dostępnie i modyfikacji zawartości zmiennych.

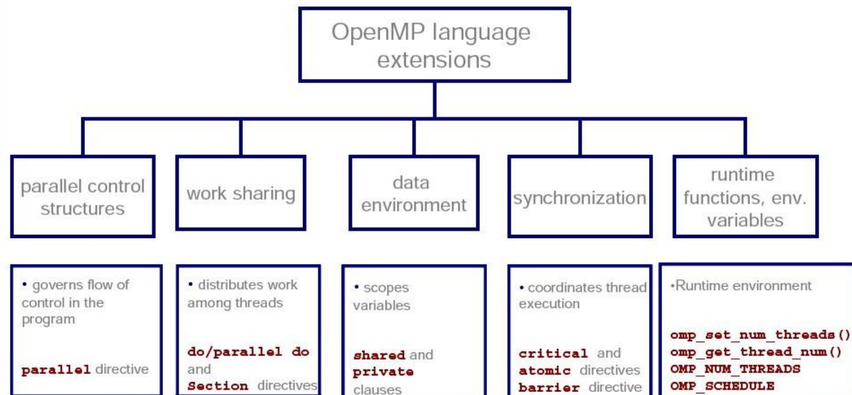
Synchronizacja

Synchronizacja

Komunikacja między wątkami odbywa się przez operacje na zmiennych współdzielonych. Synchronizacja pozwala uniknąć konfliktów w dostępnie i modyfikacji zawartości zmiennych.

- `critical` – tylko jeden wątek posiada dostęp do bloku kodu na czas jego wykonywania (*mutual exclusion*).
- `barrier` – każdy wątek czeka, aż wszystkie pozostałe dotrą do danej instrukcji w kodzie. Dopiero wówczas kolejna linia kodu zostaje wykonana (*implicit* lub *explicit*).

Wprowadzenie



Czy poniższy program można zrównoleglić w OpenMP?

```
1  program saxpy
2    use omp_lib
3    implicit none
4
5    integer ::  i, iam
6    real, dimension(10) :: z, x
7    real :: a = 2
8    real :: y = 3
9
10   call random_number(x)
11
12   do i=1, size(x)
13      z(i) = a * x(i) + y
14      write(*,*) z(i)
15   end do
16 end program saxpy
```

Czy poniższy program można zrównoleglić w OpenMP?

```fortran
program saxpy
  use omp_lib
  implicit none

  integer ::  i, iam
  real, dimension(10) :: z, x
  real :: a = 2
  real :: y = 3

  call random_number(x)

  do i=1, size(x)
     z(i) = a * x(i) + y
     write(*,*) z(i)
  end do
end program saxpy
```

```fortran
!$omp parallel do
```

Czy poniższy program można zrównoleglić w OpenMP?

```
1 program saxpy
2   use omp_lib
3   implicit none
4
5   integer ::  i, iam
6   real, dimension(10) :: z, x
7   real :: a = 2
8   real :: y = 3
9
10  call random_number(x)
11
12  do i=1, size(x)
13     z(i) = a * x(i) + y
14     write(*,*) z(i)
15  end do
16 end program saxpy
```

```
1 !$omp parallel do
```

- Iteracje pętli są dzielone na podzbiory rozdzielane wątkom w zespole (*team*),

Czy poniższy program można zrównoleglić w OpenMP?

```fortran
program saxpy
  use omp_lib
  implicit none

  integer ::  i, iam
  real, dimension(10) :: z, x
  real :: a = 2
  real :: y = 3

  call random_number(x)

  do i=1, size(x)
     z(i) = a * x(i) + y
     write(*,*) z(i)
  end do
end program saxpy
```

```fortran
!$omp parallel do
```

- Iteracje pętli są dzielone na podzbiory rozdzielane wątkom w zespole (*team*),
- Co z dyrektywą `barrier`?

Czy poniższy program można zrównoleglić w OpenMP?

```
program saxpy
  use omp_lib
  implicit none

  integer ::  i, iam
  real, dimension(10) :: z, x
  real :: a = 2
  real :: y = 3

  call random_number(x)

  do i=1, size(x)
     z(i) = a * x(i) + y
     write(*,*) z(i)
  end do
end program saxpy
```

```
!$omp parallel do
```

- Iteracje pętli są dzielone na podzbiory rozdzielane wątkom w zespole (*team*),
- Co z dyrektywą `barrier`?
- *Implicit barrier*! Każdy wątek czeka na pozostałe.

Czy poniższy program można zrównoleglić w OpenMP?

```fortran
program saxpy
  use omp_lib
  implicit none

  integer ::  i, iam
  real, dimension(10) :: z, x
  real :: a = 2
  real :: y = 3

  call random_number(x)

  do i=1, size(x)
     z(i) = a * x(i) + y
     write(*,*) z(i)
  end do
end program saxpy
```

```fortran
!$omp parallel do
```

- Iteracje pętli są dzielone na podzbiory rozdzielane wątkom w zespole (*team*),
- Co z dyrektywą `barrier`?
- *Implicit barrier*! Każdy wątek czeka na pozostałe.
- Jaki jest zasięg zmiennych? Czy program na pewno działa poprawnie?

Czy poniższy program można zrównoleglić w OpenMP?

```
1  program saxpy
2    use omp_lib
3    implicit none
4
5    integer ::  i, iam
6    real, dimension(10) :: z, x
7    real :: a = 2
8    real :: y = 3
9
10   call random_number(x)
11
12   do i=1, size(x)
13      z(i) = a * x(i) + y
14      write(*,*) z(i)
15   end do
16 end program saxpy
```

```
1 !$omp parallel do
```

- Iteracje pętli są dzielone na podzbiory rozdzielane wątkom w zespole (*team*),
- Co z dyrektywą `barrier`?
- *Implicit barrier*! Każdy wątek czeka na pozostałe.
- Jaki jest zasięg zmiennych? Czy program na pewno działa poprawnie? Indeks `i` zawsze ma status `private`!

Co z synchronizacją w programie *saxpy*?

- `barrier` – niejawnie na końcu pętli `do`.

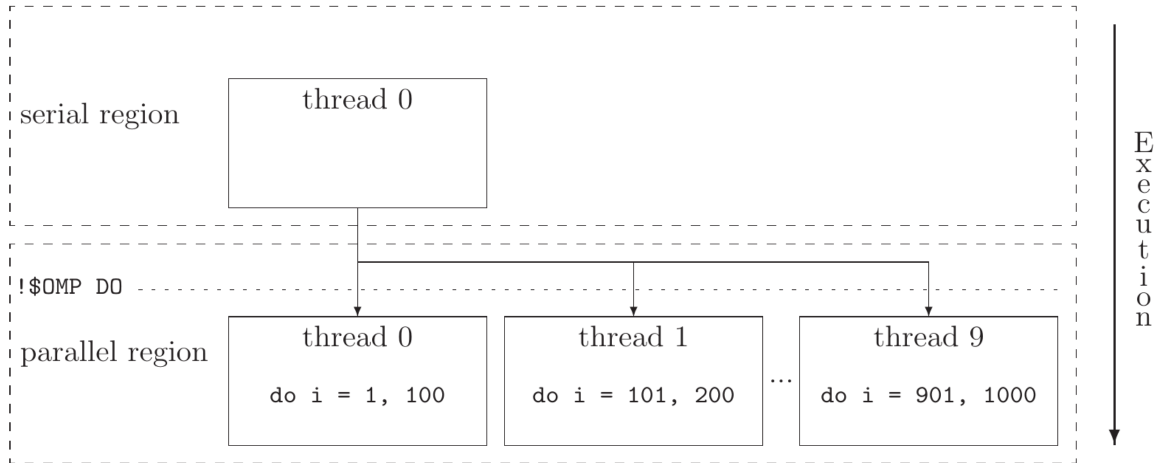
Co z synchronizacją w programie *saxpy*?

- `barrier` – niejawnie na końcu pętli `do`.
- Czy potrzebna jest synchronizacja w dostępnie do elementów tablicy `z(i)`?

Co z synchronizacją w programie *saxpy*?

- `barrier` – niejawnie na końcu pętli `do`.
- Czy potrzebna jest synchronizacja w dostępnie do elementów tablicy `z(i)`?
- Zrównoleglanie na poziomie pętli jest proste, ale efektywność tego podejścia zależy od szczegółów programu.

Pętla do



Pętla `do`

Pętle `do` (Fortran) i `for` (C/C++) to te fragmenty programów, których wykonanie (w typowych aplikacjach dziedzinowych) trwa najdłużej.

Pętla do

Pętle `do` (Fortran) i `for` (C/C++) to te fragmenty programów, których wykonanie (w typowych aplikacjach dziedzinowych) trwa najdłużej.

```
1 !$omp parallel do [clause [,] [clause...]]
2 do index = first, last [. stride]
3 body of the loop
4 enddo
5 [!omp end parallel do]
```

- Dyrektywa `parallel do` musi wystąpić bezpośrednio przed pętlą `do`;

Pętla do

Pętle do (Fortran) i for (C/C++) to te fragmenty programów, których wykonanie (w typowych aplikacjach dziedzinowych) trwa najdłużej.

```
1 !$omp parallel do [clause [,] [clause...]]
2 do index = first, last [. stride]
3 body of the loop
4 enddo
5 [!omp end parallel do]
```

- Dyrektywa parallel do musi wystąpić bezpośrednio przed pętlą do;
- Klauzule shared i private;

Pętla do

Pętle do (Fortran) i for (C/C++) to te fragmenty programów, których wykonanie (w typowych aplikacjach dziedzinowych) trwa najdłużej.

```
1 !$omp parallel do [clause [,] [clause...]]
2 do index = first, last [. stride]
3 body of the loop
4 enddo
5 [!omp end parallel do]
```

- Dyrektywa parallel do musi wystąpić bezpośrednio przed pętlą do;
- Klauzule shared i private;
- Klauzula schedule – kontroluje sposób dystrybucji zadań między wątki;

Pętla do

Pętle do (Fortran) i for (C/C++) to te fragmenty programów, których wykonanie (w typowych aplikacjach dziedzinowych) trwa najdłużej.

```
1 !$omp parallel do [clause [,] [clause...]]
2 do index = first, last [. stride]
3 body of the loop
4 enddo
5 [!omp end parallel do]
```

- Dyrektywa parallel do musi wystąpić bezpośrednio przed pętlą do;
- Klauzule shared i private;
- Klauzula schedule – kontroluje sposób dystrybucji zadań między wątki;
- Klauzula ordered – wpływa na kolejność wykonywania operacji,

Pętla do

Pętle do (Fortran) i for (C/C++) to te fragmenty programów, których wykonanie (w typowych aplikacjach dziedzinowych) trwa najdłużej.

```
1 !$omp parallel do [clause [,] [clause...]]
2 do index = first, last [. stride]
3 body of the loop
4 enddo
5 [!omp end parallel do]
```

- Dyrektywa parallel do musi wystąpić bezpośrednio przed pętlą do;
- Klauzule shared i private;
- Klauzula schedule – kontroluje sposób dystrybucji zadań między wątki;
- Klauzula ordered – wpływa na kolejność wykonywania operacji,
- Synchronizacja nie jest konieczna (niezależne elementy z(i)),

Pętla do

Pętle do (Fortran) i for (C/C++) to te fragmenty programów, których wykonanie (w typowych aplikacjach dziedzinowych) trwa najdłużej.

```
1 !$omp parallel do [clause [,] [clause...]]
2 do index = first, last [. stride]
3 body of the loop
4 enddo
5 [!omp end parallel do]
```

- Dyrektywa `parallel do` musi wystąpić bezpośrednio przed pętlą do;
- Klauzule `shared` i `private`;
- Klauzula `schedule` – kontroluje sposób dystrybucji zadań między wątki;
- Klauzula `ordered` – wpływa na kolejność wykonywania operacji,
- Synchronizacja nie jest konieczna (niezależne elementy `z(i)`),
- Bariera jest konieczna (*implicit*)!

Pętla do

- Liczba iteracji musi być znana przed jej wykonaniem;
- Pętla nie może zawierać zadnych wyrażeń typu `goto`, (możliwy jest `cycle`);
- Synchronizacja (*join*) może obniżać wydajność przy dużej ilości obszarów `parallel` (szybkość ograniczona przez najwolniejszy z wątków).

Pętla do

- Liczba iteracji musi być znana przed jej wykonaniem;
- Pętla nie może zawierać zadnych wyrażeń typu `goto`, (możliwy jest `cycle`);
- Synchronizacja (*join*) może obniżać wydajność przy dużej ilości obszarów `parallel` (szybkość ograniczona przez najwolniejszy z wątków).

Pętla może być *zagnieżdżona* (nested loop). Jeżeli dyrektywa `parallel do` jest użyta wewnątrz takiego *gniazda*, to odnosi się ona wyłącznie do tej pętli, która występuje bezpośrednio po niej.

Pętla `do`

- Liczba iteracji musi być znana przed jej wykonaniem;
- Pętla nie może zawierać zadnych wyrażeń typu `goto`, (możliwy jest `cycle`);
- Synchronizacja (*join*) może obniżać wydajność przy dużej ilości obszarów `parallel` (szybkość ograniczona przez najwolniejszy z wątków).

Pętla może być *zagnieżdżona* (nested loop). Jeżeli dyrektywa `parallel do` jest użyta wewnątrz takiego *gniazda*, to odnosi się ona wyłącznie do tej pętli, która występuje bezpośrednio po niej. (Co z zasięgiem zmiennych?)

Pętla do

- Liczba iteracji musi być znana przed jej wykonaniem;
- Pętla nie może zawierać zadnych wyrażeń typu `goto`, (możliwy jest `cycle`);
- Synchronizacja (*join*) może obniżać wydajność przy dużej ilości obszarów `parallel` (szybkość ograniczona przez najwolniejszy z wątków).

Pętla może być *zagnieżdżona* (nested loop). Jeżeli dyrektywa `parallel do` jest użyta wewnątrz takiego *gniazda*, to odnosi się ona wyłącznie do tej pętli, która występuje bezpośrednio po niej. (Co z zasięgiem zmiennych?)

```
1 do i = 1, 10
2   do j = 1, 10
3    do k = 1, 10
4     (body of the loop)
5    enddo
6   enddo
7 enddo
```

Dyrektywy `SECTIONS` i `SECTION`

Wykonanie w modelu MPMD (*Multiple Programs Multiple Data*): dany blok kodu jest wykonywany wyłącznie raz przez wątek w danym zespole.

Dyrektywy SECTIONS i SECTION

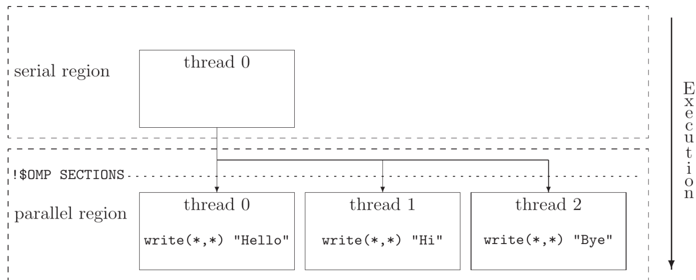
Wykonanie w modelu MPMD (*Multiple Programs Multiple Data*): dany blok kodu jest wykonywany wyłącznie raz przez wątek w danym zespole.

```
1 !$OMP SECTIONS
2 !$OMP SECTION
3 write(*,*) ``Hello''
4 !$OMP SECTION
5 write(*,*) ``Hi''
6 !$OMP SECTION
7 write(*,*) ``Bye''
8 !$OMP END SECTIONS
```

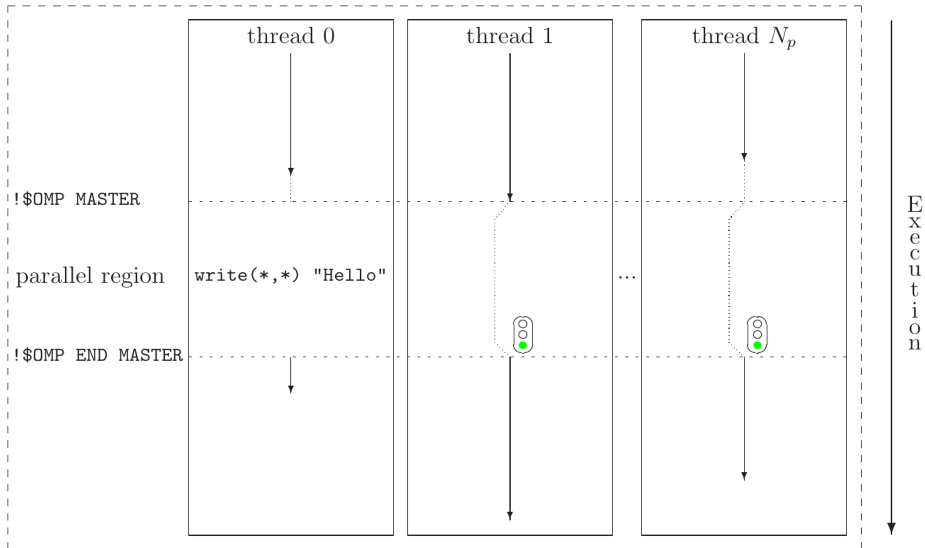
Dyrektywy SECTIONS i SECTION

Wykonanie w modelu MPMD (*Multiple Programs Multiple Data*): dany blok kodu jest wykonywany wyłącznie raz przez wątek w danym zespole.

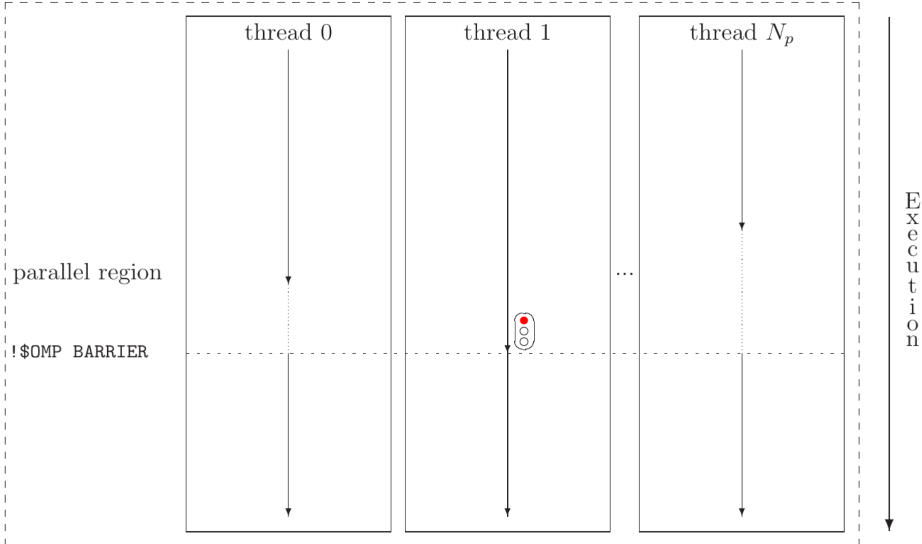
```
1 !$OMP SECTIONS
2 !$OMP SECTION
3 write(*,*) ''Hello''
4 !$OMP SECTION
5 write(*,*) ''Hi''
6 !$OMP SECTION
7 write(*,*) ''Bye''
8 !$OMP END SECTIONS
```



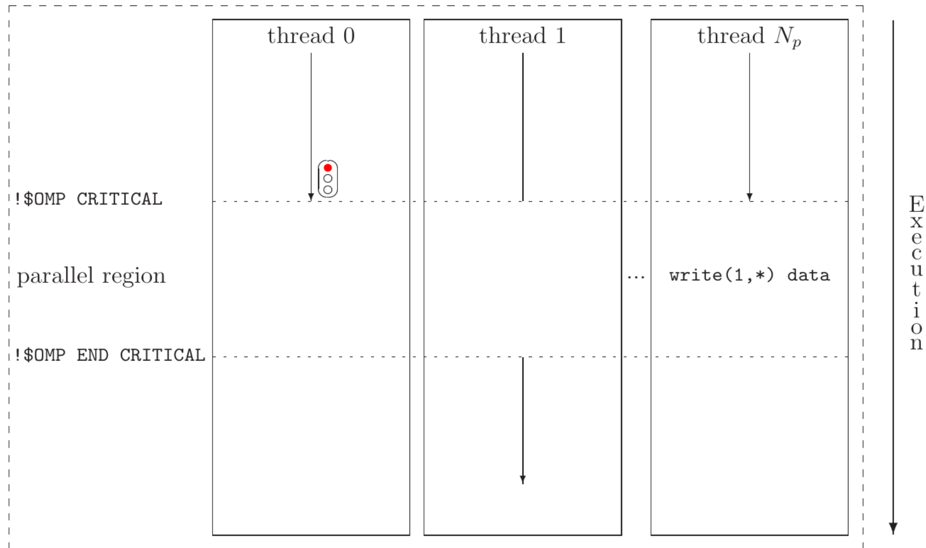
Dyrektywa MASTER



Dyrektywa BARRIER



Dyrektywa CRITICAL



Tasking

`task`:

Defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct, per-data environment ICVs, and any defaults that apply.

Tasking

task:

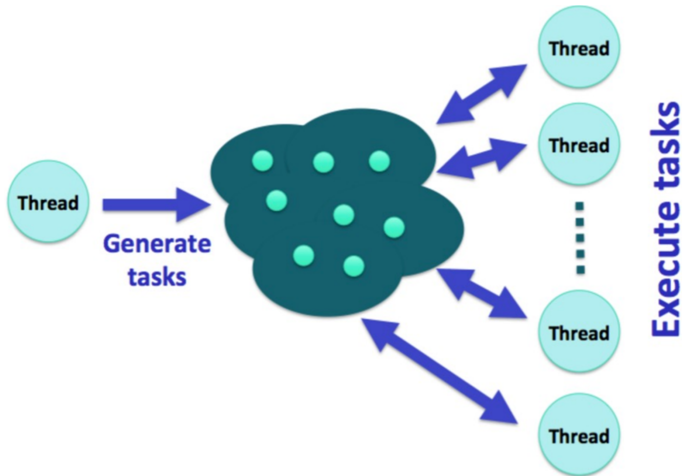
Defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct, per-data environment ICVs, and any defaults that apply.

```
!$omp task [clause[ [, ]clause] ... ]
  loosely-structured-block
!$omp end task
```

Tasking vs. Sections

- Sekcje definiowane wewnątrz bloku `sections`
- Na końcu sekcji synchronizacja (`barrier`) (wątki muszą czekać na zakończenie bloku)
- `taskwait` – wątki czekają na zakończenie wszystkich `tasks`
- `tasks` mogą być generowane przez pojedynczy wątek w sekcji `single`

Tasking



F. Desprez - UE Parallel alg. and prog.

REDUCTION

reduction

```
reduction(operator:list)
```

REDUCTION

reduction

```
reduction(operator:list)
```

Zmienna zawarta w `list` jest aktualizowana poprzez działanie operatora. Dla każdego wątku zostaje utworzona kopia tej zmiennej o statusie `private`. Po zakończeniu pracy przez wątki, poszczególne wyniki są zbierane do współdzielonej zmiennej.

REDUCTION

```
do i = 1, n
   sum = sum + a(i)
enddo
```

```
for(i=1; i<=n; i++) {
   sum = sum + a[i];
}
```

- Jaki powinien być zasięg zmiennej `sum`?

REDUCTION

```
do i = 1, n
   sum = sum + a(i)
enddo
```

```
for(i=1; i<=n; i++) {
   sum = sum + a[i];
}
```

- Jaki powinien być zasięg zmiennej `sum`?
- Operacja redukcji zmiennej,

REDUCTION

```
do i = 1, n
   sum = sum + a(i)
enddo
```

```
for(i=1; i<=n; i++) {
   sum = sum + a[i];
}
```

- Jaki powinien być zasięg zmiennej `sum`?
- Operacja redukcji zmiennej,
- Klauzula: `reduction(operator:list)`.

Wydajność aplikacji OpenMP

- *Parallel overhead* – koszt zrównoleglania;
  - fork/join/synchronizacja,
  - dystrybucja zadań pomiędzy wątki (podział iteracji pętli).

Wydajność aplikacji OpenMP

- *Parallel overhead* – koszt zrównoleglania;
  - fork/join/synchronizacja,
  - dystrybucja zadań pomiędzy wątki (podział iteracji pętli).

- Co jeżeli iteracje pętli wykonują zadania o różnym stopniu złożoności?

Wydajność aplikacji OpenMP

- *Parallel overhead* – koszt zrównoleglania;
  - fork/join/synchronizacja,
  - dystrybucja zadań pomiędzy wątki (podział iteracji pętli).

- Co jeżeli iteracje pętli wykonują zadania o różnym stopniu złożoności?
- Synchronizacja spowoduje opóźnienia w wykonaniu równoległym (niektóre wątki będą pracowały dłużej niż inne);

Wydajność aplikacji OpenMP

- *Parallel overhead* – koszt zrównoleglania;
  - fork/join/synchronizacja,
  - dystrybucja zadań pomiędzy wątki (podział iteracji pętli).

- Co jeżeli iteracje pętli wykonują zadania o różnym stopniu złożoności?
- Synchronizacja spowoduje opóźnienia w wykonaniu równoległym (niektóre wątki będą pracowały dłużej niż inne);
- Klauzula `schedule` (dyrektywa `parallel do`) umożliwia zbalansowanie obciążenia.

schedule

SCHEDULE

– sposób dystrybucji zadań pomiędzy dostępne wątki

schedule

SCHEDULE

– sposób dystrybucji zadań pomiędzy dostępne wątki

```
schedule ([modifier] [, chunk_size])
```

schedule

SCHEDULE

– sposób dystrybucji zadań pomiędzy dostępne wątki

```
schedule ([modifier] [, chunk_size])
```

modifier

`static`: *Iterations are divided into chunks of size chunk_size and assigned to team threads in round-robin fashion in order of thread number.*

`dynamic`: *Each thread executes a chunk of iterations,* ***then*** *requests another chunk until none remain.*

`guided`: *Same as dynamic, except chunk size is different for each chunk, with each successive chunk smaller than the last.*

`schedule`

Zalety i wady modyfikatorów klauzuli `schedule`

- `dynamic` wymusza koordynację, żeby każda iteracja wykonała się jeden raz $\rightarrow$ **koszt**;
- `static` nie ponosi tego kosztu, ale traci na tym, jeżeli iteracje wykonują bardzo zróżnicowane zadania;

`schedule`

Zalety i wady modyfikatorów klauzuli `schedule`

- `dynamic` wymusza koordynację, żeby każda iteracja wykonała się jeden raz $\rightarrow$ **koszt**;
- `static` nie ponosi tego kosztu, ale traci na tym, jeżeli iteracje wykonują bardzo zróżnicowane zadania;

- Przy statycznym podziale zadań, każdy wątek otrzymuje równy co do wielkości `chunk` (porcję) pracy do wykonania (jeżeli liczba iteracji nie dzieli się całkowicie na liczbę wątków, to o podziale decyduje implementacja OpenMP);

`schedule`

Zalety i wady modyfikatorów klauzuli `schedule`

- `dynamic` wymusza koordynację, żeby każda iteracja wykonała się jeden raz $\rightarrow$ **koszt**;
- `static` nie ponosi tego kosztu, ale traci na tym, jeżeli iteracje wykonują bardzo zróżnicowane zadania;

- Przy statycznym podziale zadań, każdy wątek otrzymuje równy co do wielkości `chunk` (porcję) pracy do wykonania (jeżeli liczba iteracji nie dzieli się całkowicie na liczbę wątków, to o podziale decyduje implementacja OpenMP);
- Przy dynamicznym podziale – domyślny `chunk` jest równy 1;

schedule

Zalety i wady modyfikatorów klauzuli schedule

- dynamic wymusza koordynację, żeby każda iteracja wykonała się jeden raz $\rightarrow$ **koszt**;
- static nie ponosi tego kosztu, ale traci na tym, jeżeli iteracje wykonują bardzo zróżnicowane zadania;

- Przy statycznym podziale zadań, każdy wątek otrzymuje równy co do wielkości chunk (porcję) pracy do wykonania (jeżeli liczba iteracji nie dzieli się całkowicie na liczbę wątków, to o podziale decyduje implementacja OpenMP);
- Przy dynamicznym podziale – domyślny chunk jest równy 1;
- guided – chunk jest zmienny;

`schedule`

Zalety i wady modyfikatorów klauzuli `schedule`

- `dynamic` wymusza koordynację, żeby każda iteracja wykonała się jeden raz → **koszt**;
- `static` nie ponosi tego kosztu, ale traci na tym, jeżeli iteracje wykonują bardzo zróżnicowane zadania;

- Przy statycznym podziale zadań, każdy wątek otrzymuje równy co do wielkości `chunk` (porcję) pracy do wykonania (jeżeli liczba iteracji nie dzieli się całkowicie na liczbę wątków, to o podziale decyduje implementacja OpenMP);
- Przy dynamicznym podziale – domyślny `chunk` jest równy 1;
- `guided` – `chunk` jest zmienny;
- `runtime` – określony przez zmienną środowiskową:

schedule

Zalety i wady modyfikatorów klauzuli schedule

- `dynamic` wymusza koordynację, żeby każda iteracja wykonała się jeden raz → **koszt**;
- `static` nie ponosi tego kosztu, ale traci na tym, jeżeli iteracje wykonują bardzo zróżnicowane zadania;

- Przy statycznym podziale zadań, każdy wątek otrzymuje równy co do wielkości `chunk` (porcję) pracy do wykonania (jeżeli liczba iteracji nie dzieli się całkowicie na liczbę wątków, to o podziale decyduje implementacja OpenMP);
- Przy dynamicznym podziale – domyślny `chunk` jest równy 1;
- `guided` – `chunk` jest zmienny;
- `runtime` – określony przez zmienną środowiskową:

```
setenv OMP_SCHEDULE 'dynamic,3'
```

Wybrane funkcje OpenMP

`omp_get_num_threads()`
`omp_get_thread_num()`

Zmienna `OMP_NUM_THREADS`

Sesja praktyczna

https://ondemand.hpc.icm.edu.pl

```
git clone https://git.icm.edu.pl/mszpindl/computing.git
```