Please download the latest slides from obe.

OpenMP Basics

Zhang Feng

2022-9-30

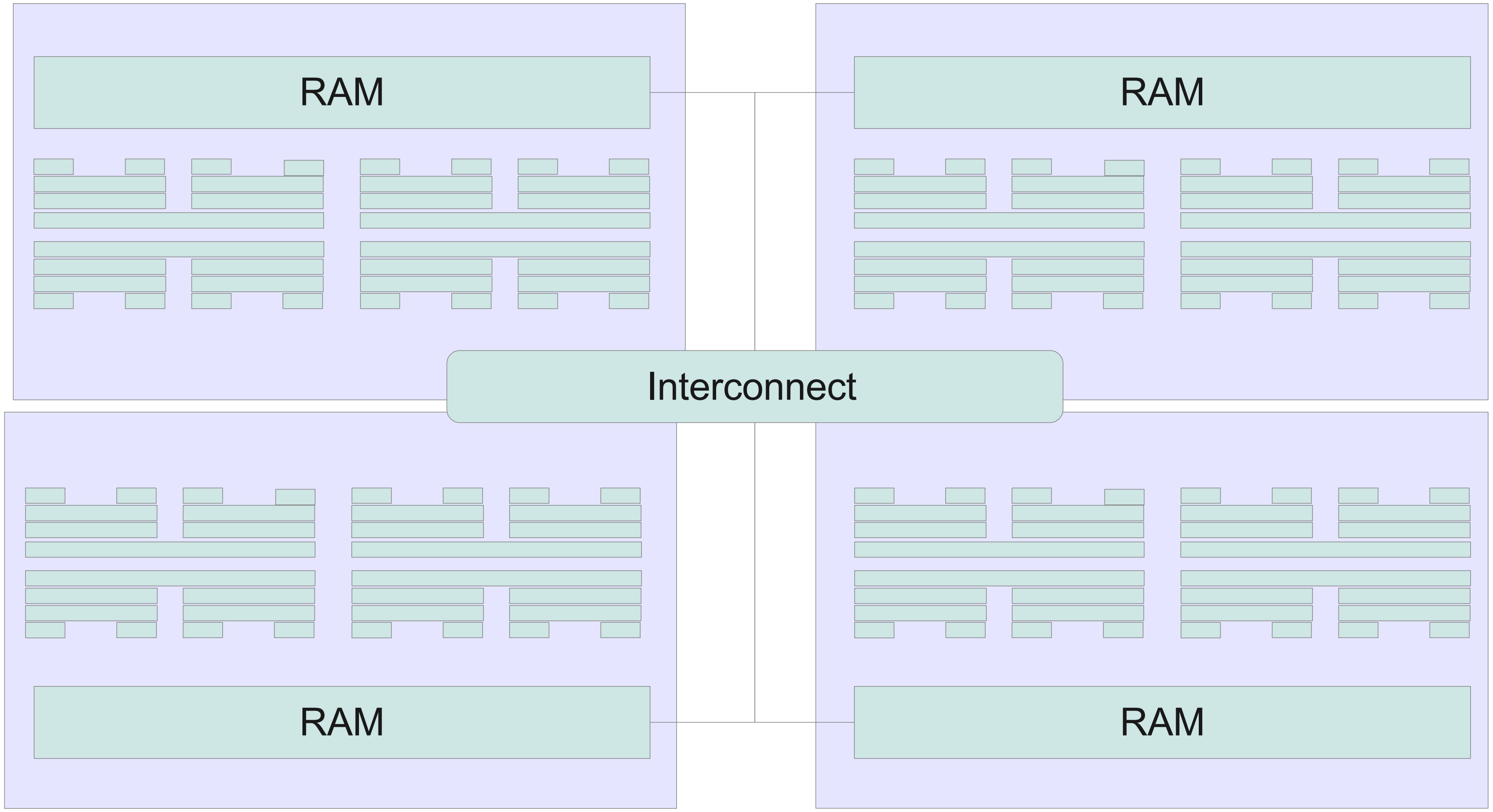
Outline

- Parallel Architectures
  - SMP： Symmetric Multi-Processor
  - NUMA： Non-Uniform Memory Access
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax

Parallel Architectures

- NUMA (Non-Uniform Memory Access) architecture.
- Linking several SMPs (Symmetric Multi-Processor).
- Coherency not maintained.

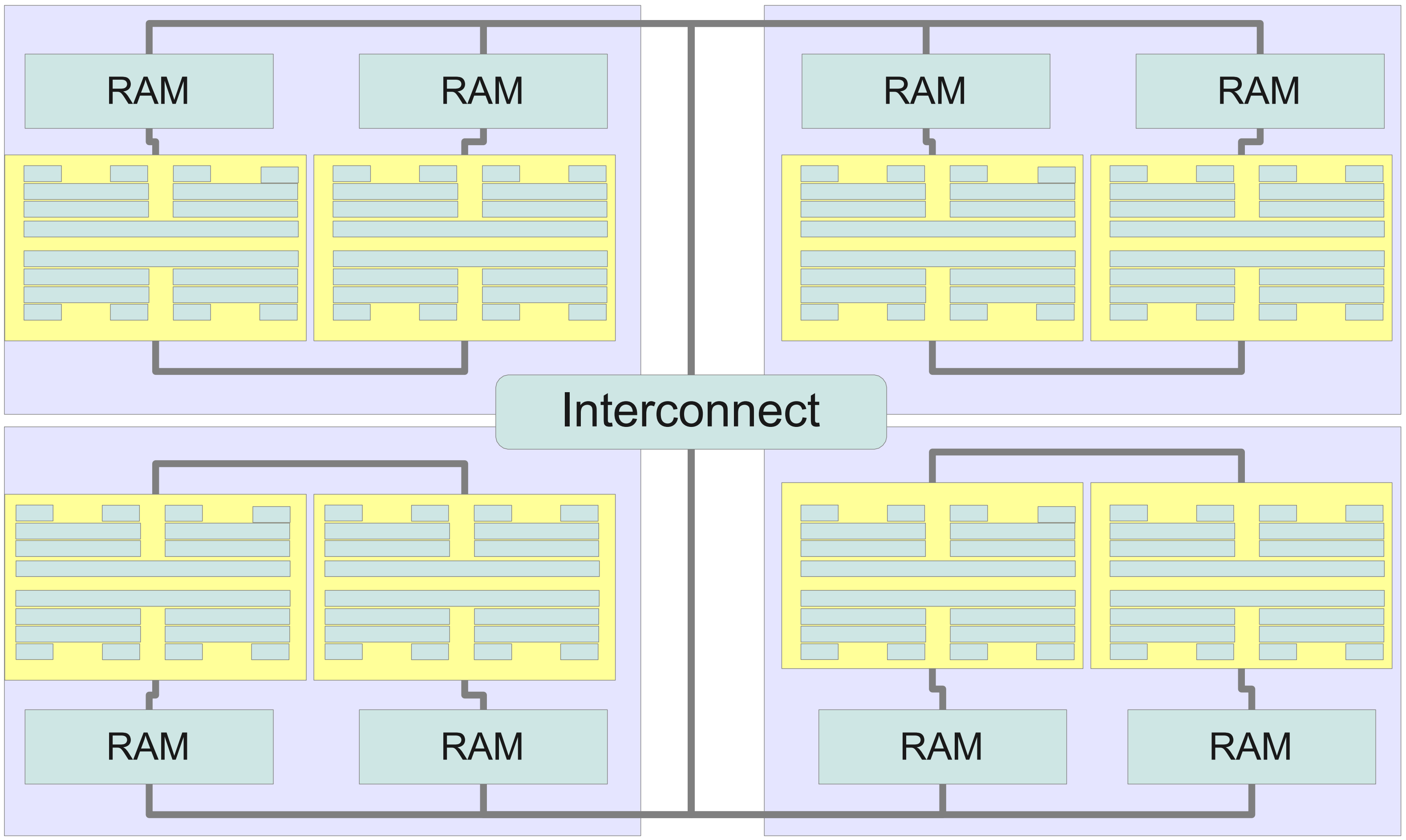
Distributed Memory



Parallel Architectures

- ccNUMA (cache coherent Non-Uniform Memory Access) architecture.
- Communication between cache controllers to maintain coherency.
- Consistent memory image.

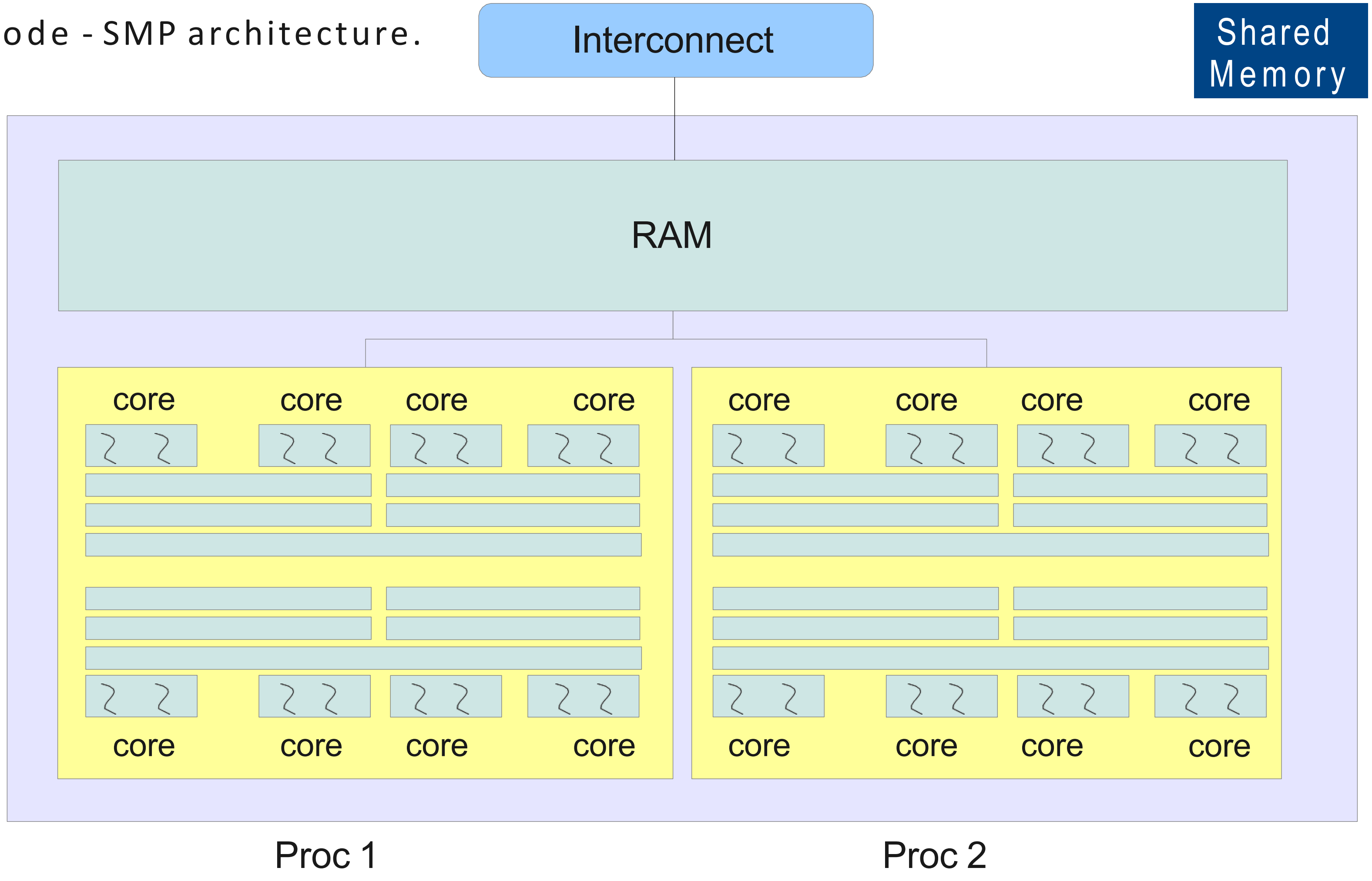
Distributed Memory



Parallel Architectures

NUMA node - SMP architecture.

Shared Memory



Outline

- Parallel Architectures
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax

Programing Models

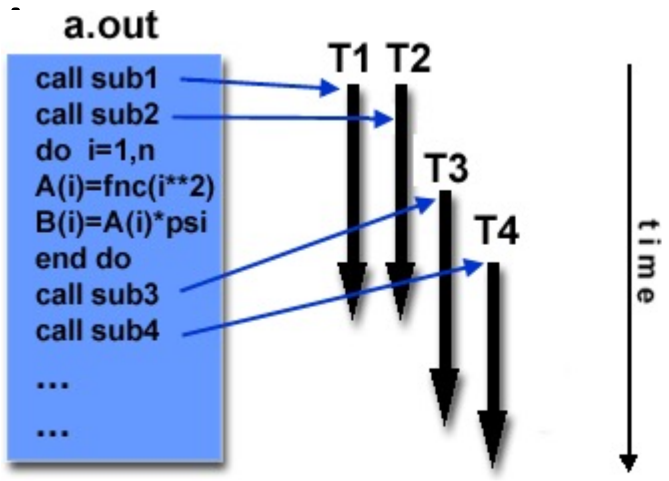
Distributed Memory
MPI

- Explicit model – message sending/receiving for：
  - Data exchange
  - Synchronization
  - Communication
- Programmer should express the parallelism explicitly.
- MPI subroutines used at source level.
- Message Passing Interface (MPI)
- The “de facto” industry standard for message passing.

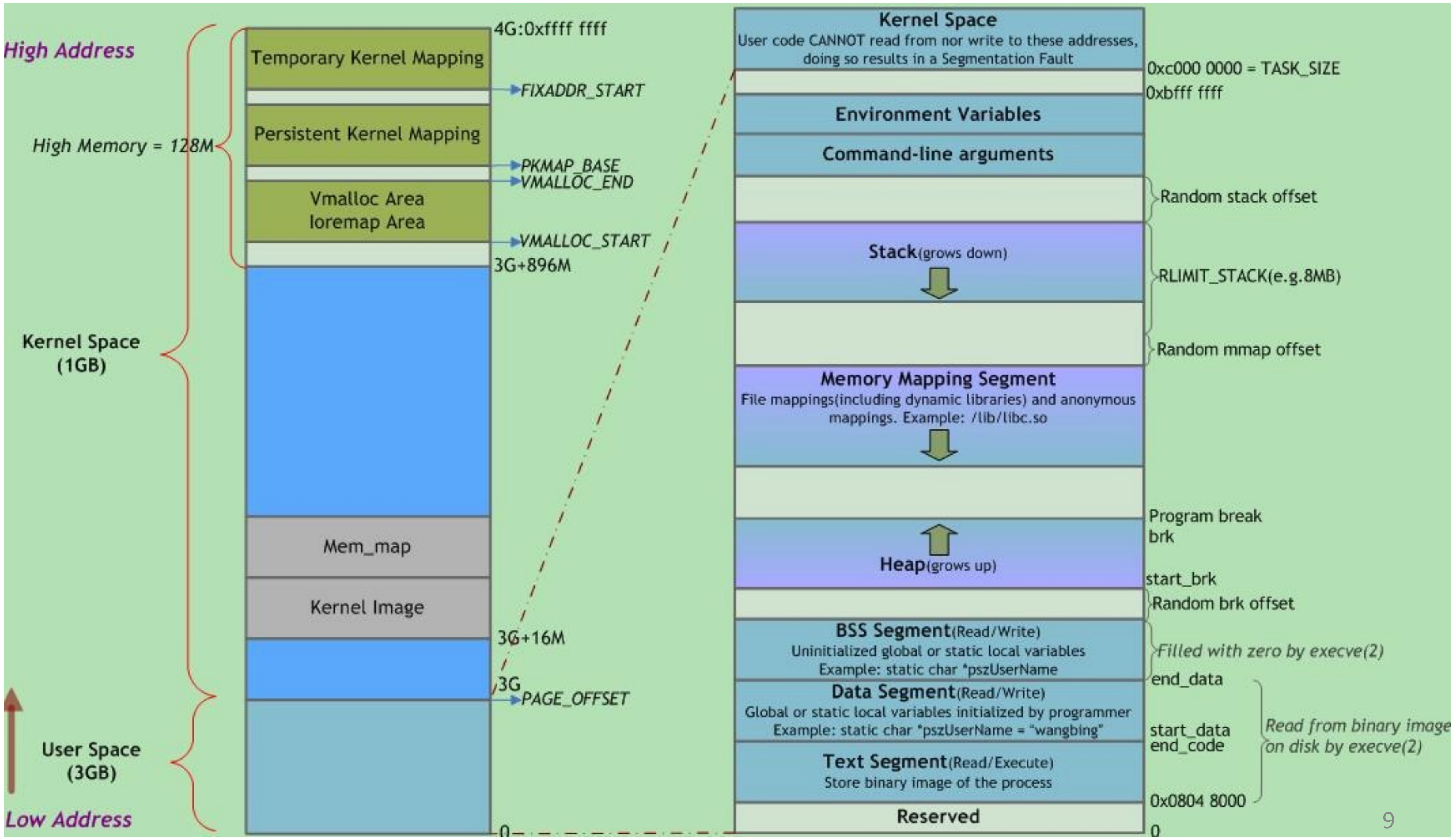
Programing Models

Shared Memory
Threads

- Thread model
- One program with multiple subroutines.
- Each thread Ti has local data.
- Each thread accesses to the global memory → potential synchronization.



LINUX process memory layout



LINUX process memory layout

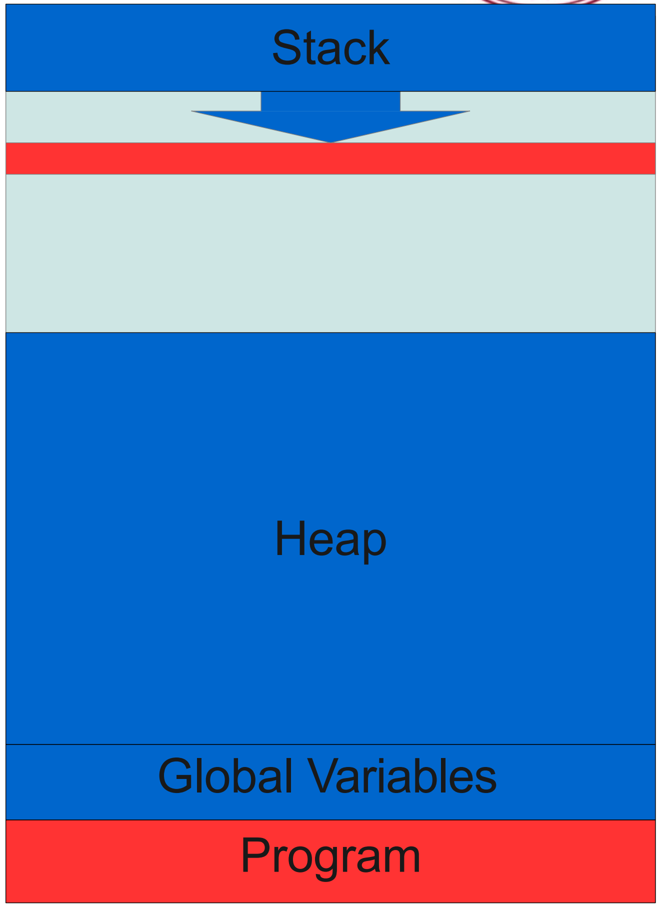
- Stack
  - local variables, function parameters, return addresses, etc.
- Heap
  - dynamically allocated memory,malloc(),new()…
- BSS segment
  - global variables and static local variables that are not initialized or initialized to 0
- Data segment
  - global variables and static local variables that are initialized and have a non-zero initial value
- Code snippet
  - executable code, string literals, read-only variables

Programing Models

Shared Memory
Threads

- What is a thread?
  - ✓ Smallest unit scheduled by the OS.
  - ✓ Different threads belong to one process.
  - ✓ Thread = lightweight process.

- One thread owns:
  - ✓ A stack.
  - ✓ A set of registers (ie. a context).

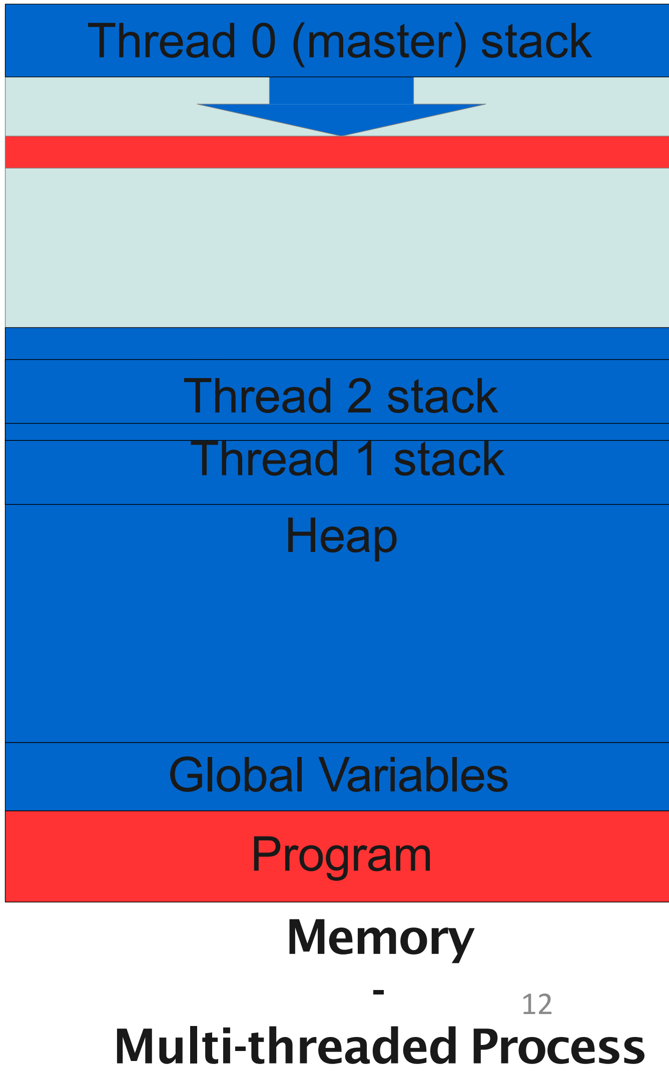


Memory
-
Sequential Process

Programing Models

Shared Memory Threads

- Multiple threads share the same @ space.
- The threads stacks are located in the heap.
- The thread's stack has a fixed size.
- Only the master thread's stack size increases.
- Global variables are shared between threads.
- Communication between threads via the memory.



Memory - Multi-threaded Process

Programing Models

Shared Memory
Threads →OpenMP

- Two APIs for kernel threads manipulation:
  - POSIX threads: Linux, FreeBSD, MacOS, Solaris, ...
  - Windows threads
- Different programing models for shared memory:
  - OpenMP, CILK, TBB, ...
- Focus on the OpenMP programing model
- OpenMP 4.5 Complete Specifications (Nov 2015)
  - http://www.openmp.org/specifications/

Question1

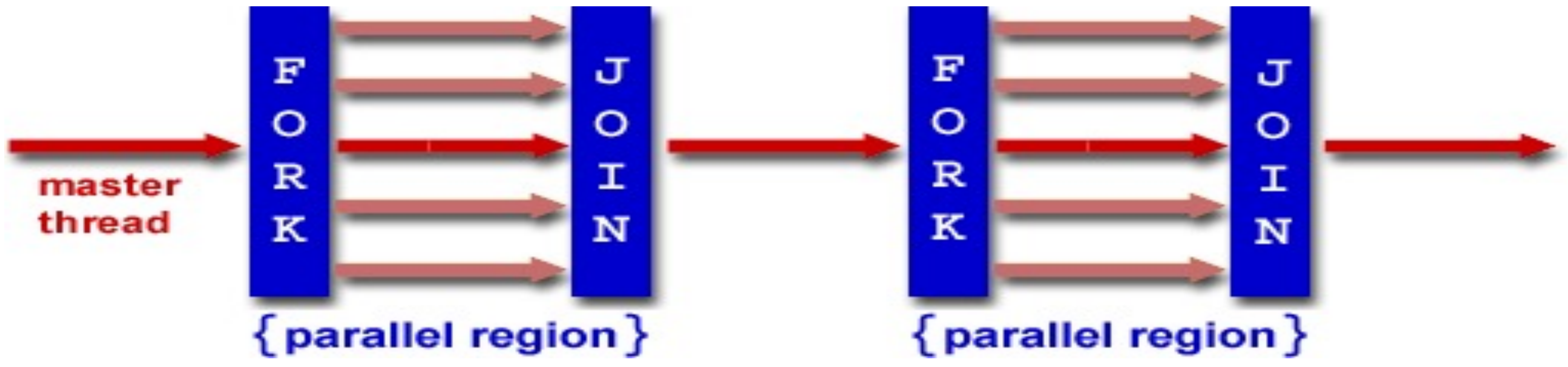
- When a process starts multiple threads, where are the stacks of these threads allocated? Can they grow?

Outline

- Parallel Architectures
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax

OpenMP

- OpenMP is a set of multi-threaded concurrent programming APIs that support cross-platform shared memory.
- OpenMP (Open Multi-Processing) is a C/C++ and Fortran Application Programming Interface for shared memory architectures.
- OpenMP is based on the Fork and Join model



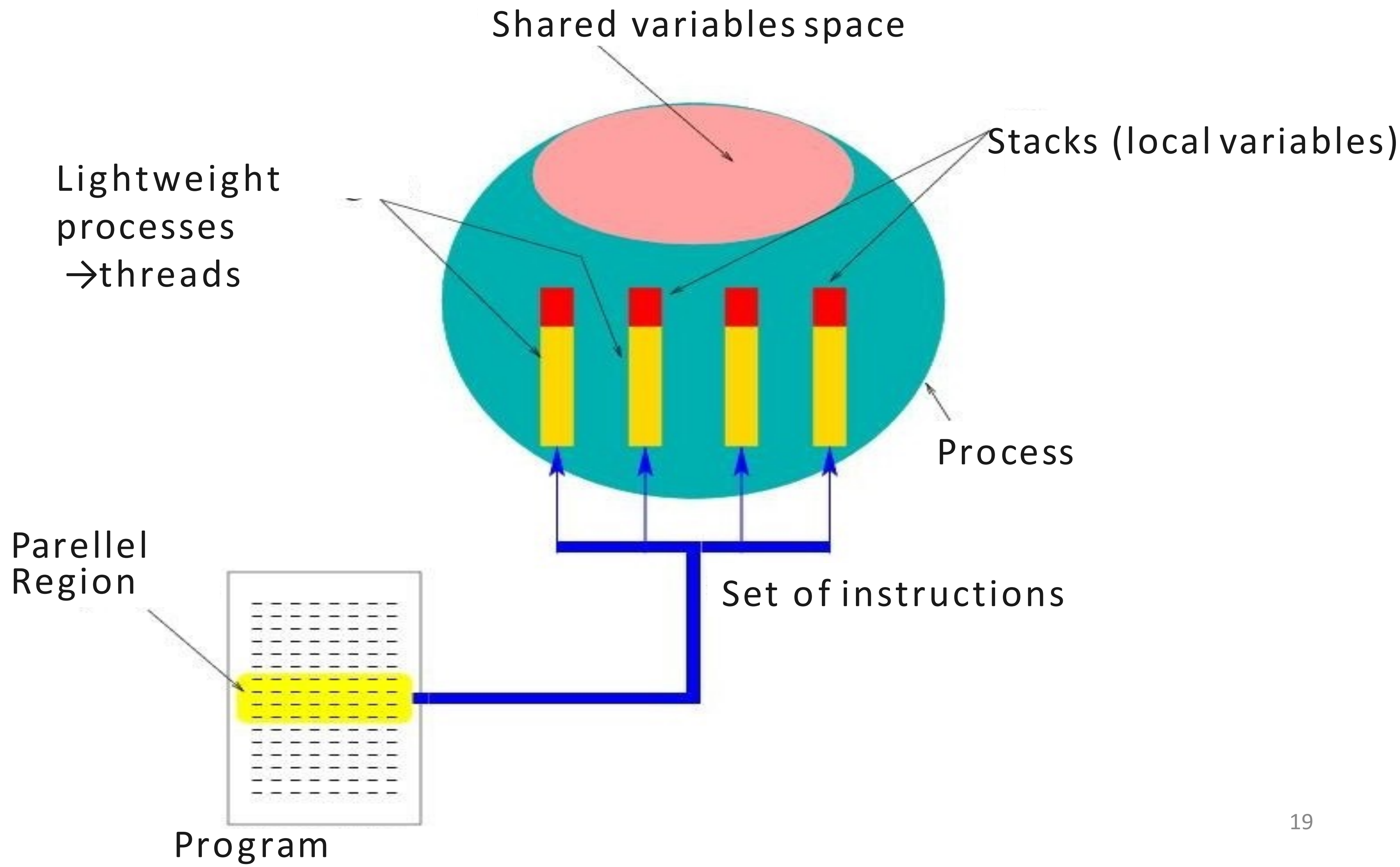
OpenMP

- OpenMP consists in:
  - ✓ A set of compiler directives.
  - ✓ Library functions calls
  - ✓ Environment variables

OpenMP

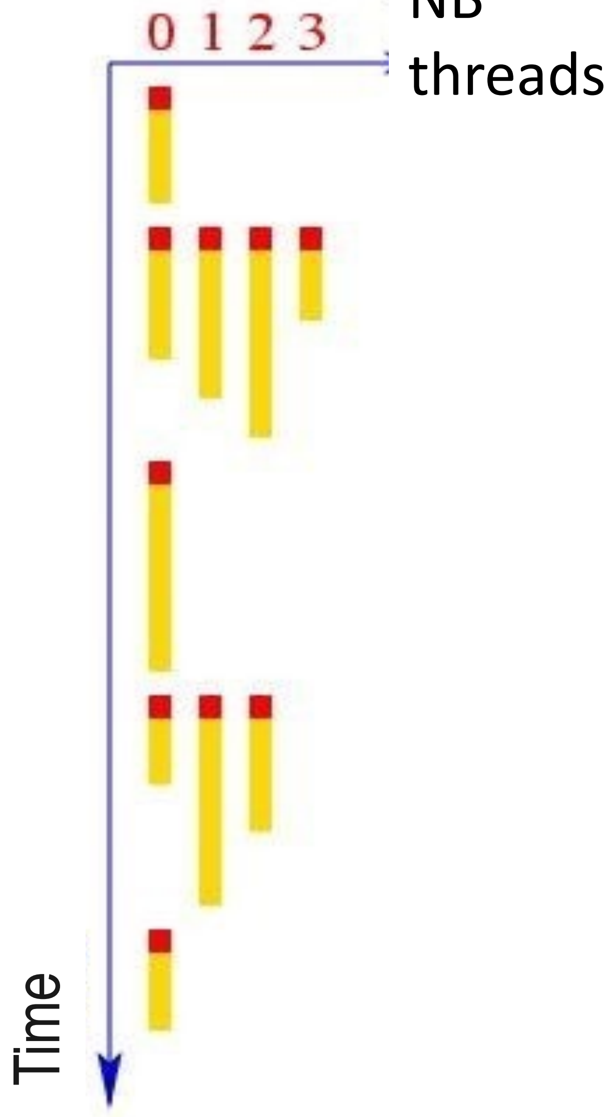
- An OpenMP program is executed by one process →thread master.
- The process activates several lightweight processes (ie. threads) when a parallel region starts →Fork.
- The code of the parallel region is duplicated and each thread executes that code.
- Different threads executes the code at the same time.
- At the end of a parallel region (ie. join), only the master thread continues execution.
- During the execution a the threads, a variable can be read/written:
  - ✓ If the variable is in the thread's stack →private variable
  - ✓ If the variable defined in a shared memory space →shared variable

OpenMP


Shared variables space
Stacks (local variables)
Lightweight processes →threads
Process
Parellel Region
Set of instructions
Program

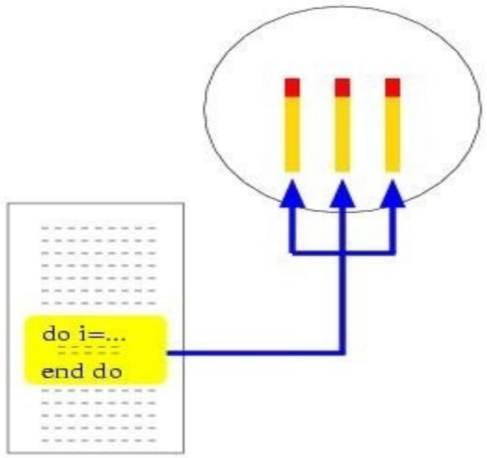
OpenMP

- An OpenMP program alternates sequential and parallel regions.
- The sequential region is always executed by the thread master →thread 0.
- A parallel region is executed by different threads at the same time.
- The threads may share the work inside the parallel region.

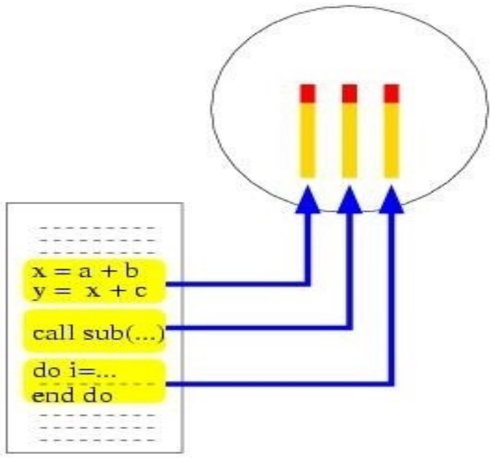


OpenMP

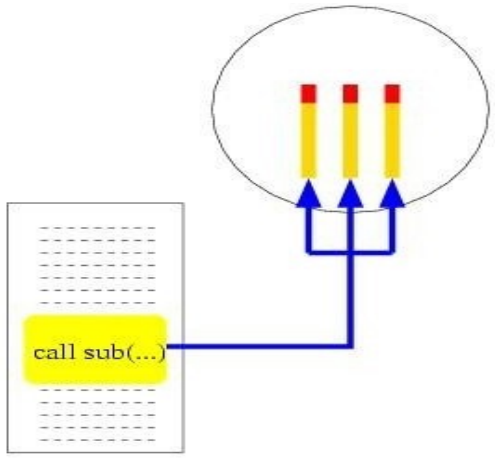
- What is the work sharing between threads?
  - ✓ Share the iteration space of a loop between threads.
  - ✓ Execute different sections of a program but one section per thread.
  - ✓ Execute different occurrences of a procedure by different threads.



Loop Level Parallelism



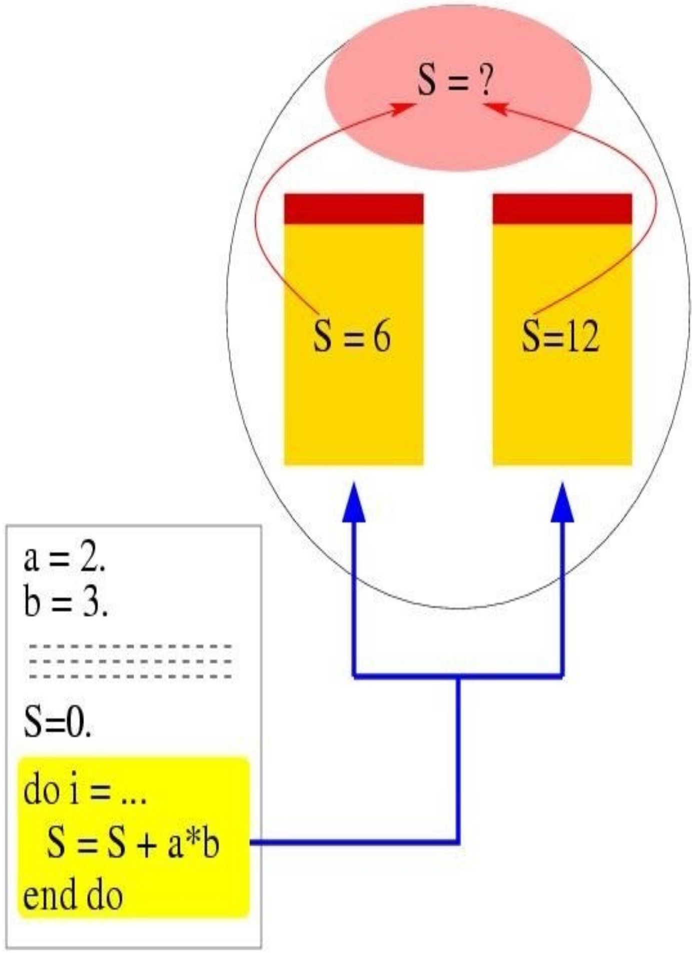
Parallel Sections



Parallel Procedures

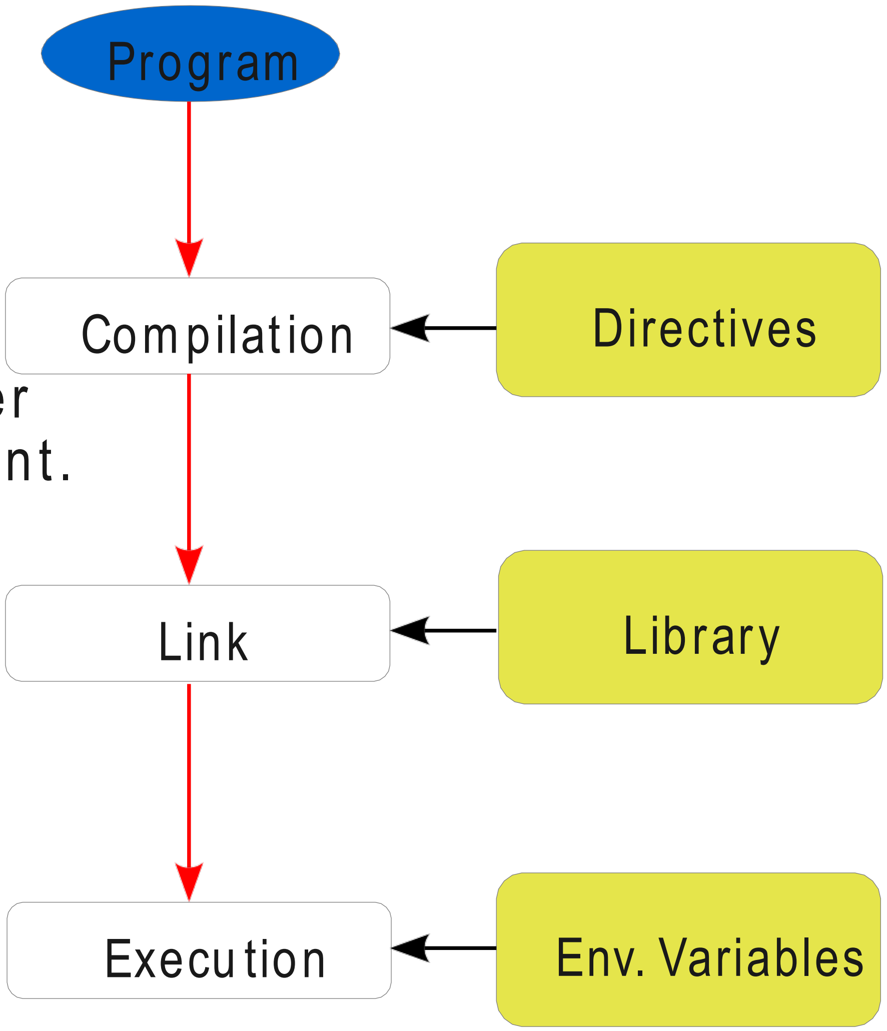
OpenMP

- Data race
- It is the access to a shared variable by different threads and at least one access is a write.
- Alleviate data races through synchronization between threads



OpenMP

- Compilation directives:
  - ✓ Define the work sharing.
  - ✓ Synchronization.
  - ✓ Privacy of variables.
  - ✓ If correct flag not set, the compiler consider the directive as a comment.
- Library functions calls:
  - ✓ It is loaded at link.
- Environment variables:
  - ✓ When set up, their values are considered at execution time.



OpenMP-Syntax

- The OpenMP directives are:
  - ✓ Inserted in the source code by the programmer
    OR
  - ✓ Inserted automatically in the source code (ie. automatic parallelization)

- An OpenMP directive has the following shape:

  #pragma omp *directive [clause[clause]...]*   *for C/C++*

  sentinel directive [clause[clause]...]   *for Fortran*

- There is an include file "*omp.h*":
  - ✓ It defines all OpenMP functions.
  - ✓ It should be included in each OpenMP program to be able to use the functions.

OpenMP- Syntax

```
#include <omp.h>

…

#pragma omp parallel private(a,b) \
              shared(d,c)
{

…

}
```

Question2

- What happens if an incorrect parallel guide statement is inserted?
- What happens if a parallel directive statement is inserted but the OpenMP library is not linked?

Outline

Parallel Region Construction

- In a parallel region by default the variables are shared.
- In a parallel region all threads execute the same code.
- At the end of the parallel region there is an implicit barrier for synchronization.
- No branches inside or outside a parallel region, neither in any other OpenMP construct.

Parallel Region Construction

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

int main(void)
{
   int a;         int   p;
   a = 10000;     p = 0;
   #pragma omp parallel
   {
#ifdef _OPENMP
      p=omp_in_parallel();
#endif
      printf("a = %d ; p = %d\n",a,p);
   }
   return 0;
}
```

Parallel Region Construction

```
> gcc … -fopenmp -o omp omp.c
> export OMP_NUM_THREADS=8
> ./omp

a = 10000 ; p = 1
a = 10000 ; p = 1
a = 10000 ; p = 1
a = 10000 ; p = 1
a = 10000 ; p = 1
a = 10000 ; p = 1
a = 10000 ; p = 1
a = 10000 ; p = 1
```

Parallel Region Construction

- It is possible to change the default status of variables in the parallel region construct. This can be done using the clause ***DEFAULT*** .
- If a variable has a ***PRIVATE*** status, it is located in the thread's stack. Its value is undefined at the entry of a parallel region.

Parallel Region Construction

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

int main(void)
{
   int a = 10000;
   int p = 0;
   #pragma omp parallel private(a)
   {
#ifdef _OPENMP
      p=omp_in_parallel();
#endif
      printf("a = %d ; p = %d\n",a,p);
   }
   return 0;
}
```

Parallel Region Construction

```
> gcc … -fopenmp -o omp omp.c
> export OMP_NUM_THREADS=8
> ./omp

a = 0 ; p = 1
a = 0 ; p = 1
a = 0 ; p = 1
a = 0 ; p = 1
a = 32621 ; p = 1
a = 0 ; p = 1
a = 0 ; p = 1
a = 0 ; p = 1
```

Parallel Region Construction

- The clause ***FIRSPRIVATE*** forces the initialization of the private varibale to the last value it has right before the parallel region.

Parallel Region Construction

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

int main(void)
{
   int a;
   a = 50000;

   #pragma omp parallel firstprivate(a)
   {
        a += 1111;
     printf("a = %d\n",a);
   }

   printf*("After parallel region, a = %d\n",a);

   return 0;
}
```

Parallel Region Construction

```
> gcc … fopenmp o omp omp.c
> export OMP_NUM_THREADS=8
> ./omp

a = 51111
a = 51111
a = 51111
a = 51111
a = 51111
a = 51111
a = 51111
a = 51111
After parallel region, a = 50000
```

Scope of a Parallel Region

- The influence of a parallel region goes beyond the region's “lexical” scope. It is extended to the code of the subroutines called in the parallel region.

- This is called ***the dynamic scope.***

Scope of a Parallel Region

```
int main(void)
{
    void sub(void);

   #pragma omp parallel
   {
      sub();
   }
   return 0;
}
```

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

void sub(void)
{
    int p=0;

#ifdef _OPENMP
    p = omp_in_parallel();
#endif

    printf("Parallel? :
      %d\n", p) ;
}
```

Scope of a Parallel Region

```
> gcc …  fopenmp o omp omp.c   sub.c
> export OMP_NUM_THREADS=8
> ./omp

Parallel? : 1
Parallel? : 1
Parallel? : 1
Parallel? : 1
Parallel? : 1
Parallel? : 1
Parallel? : 1
Parallel? : 1
```

Scope of a Parallel Region

- In a subroutine called in a parallel region, the local variables are implicitly private to each thread and defined in its corresponding stack.

```
int main(void)
{
   void sub(void);

   #pragma omp parallel \
          default(shared)
   {
      sub();
   }
   return 0;
}
```

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

void sub(void)
{
    int a;
    a = 50000;
    a = a +
    omp_get_thread_num();
    printf("a = %d\n", a);
}
```

Scope of a Parallel Region

```
> gcc …  -fopenmp o omp omp.c  sub.c
> export OMP_NUM_THREADS=8
> ./omp

a = 50000
a = 50001
a = 50007
a = 50005
a = 50004
a = 50003
a = 50002
a = 50006
```

Question3

- Why the keyword FIRSPRIVATE is designed?

Outline

- Parallel Architectures
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax
  - Parallel Region
  - Work Sharing
  - Exclusive Execution
  - Synchronization
  - Nested Parallelism
  - API + Env. Variables
  - Common mistakes
  - More Slides

Work Sharing

- Using OpenMP functions to create a parallel region is enough.

- The programmer has to explicitly divide the work, the data and make sure not to have any data race.

- OpenMP has directives that take care of that in a good way:
  - ✓ FOR directive
  - ✓ SECTIONS directive
  - ✓ WORKSHARE directive

Work Sharing – FOR directive

- A ***for*** directive is used inside a parallel region.

- Different clauses can be used with the **for** directive:
  - ✓ ***private*** – makes the variable private

  - ✓ ***firstprivate*** – makes the variable private and assign it the last value it has right before the ***for*** region

  - ✓ ***lastprivate*** – makes the variable private and keep the value it has at the last iteration of the loop, outside the ***for*** region

  - ✓ ***reduction*** – a private copy for each variable listed is created for each thread. The reduction variable is applied to all private copies of the shared variable. The final result is written to the global shared variable.

Question1

Work Sharing – FOR directive

```
#define length 3

int main(void)
{
    int A[length];
    int B[length];
    int dot = 0;

    Init(A,B);

    #pragma omp parallel default(none) shared(A,B,dot)
    {
                int k;

                #pragma omp for reduction(+:dot)
                for (k=0 ; k<length ; ++k){
                  dot += A[k] * B[k];
                }

            }

    return 0;
}
```

Length=3
A[3]={1,2,3}
B[3]={1,2,3}
What is the result of dot at the end?

Question

```
#include<stdio.h>
#include<omp.h>
int main(){
        int a[3]={1,2,3};
        int b[3]={1,2,3};
        int dot=0;

#pragma omp parallel default(none) shared(a,b,dot)
        {
                int k;
#pragma omp for reduction(+:dot)
                for(k=0; k<3; k++){
                        printf("k=%d\n",k);
                        dot+=a[k]*b[k];
                }
        }
        printf("dot=%d\n",dot);
        return 0;
}
```

Without this instructive compile statement, what would the result be, assuming 8 threads?
14/ 112

Work Sharing – FOR directive

```
#define length 500

int main(void)
{
    int A[length];
    int B[length];
    int dot = 0;

    Init(A,B);

    #pragma omp parallel default(none) shared(A,B,dot)
    {
            int k;

            #pragma omp for reduction(+:dot)
            for (k=0 ; k<length ; ++k){
              dot += A[k] * B[k];
            }

        }

    return 0;
}
```

Work Sharing – FOR directive

- Different scheduling can be applied to the threads:
  - ✓ Static
  - ✓ Dynamic
  - ✓ Guided

- The clause ***SCHEDULE*** allows to choose the scheduling:

  #pragma omp ***for schedule(static|dynamic|guided)***

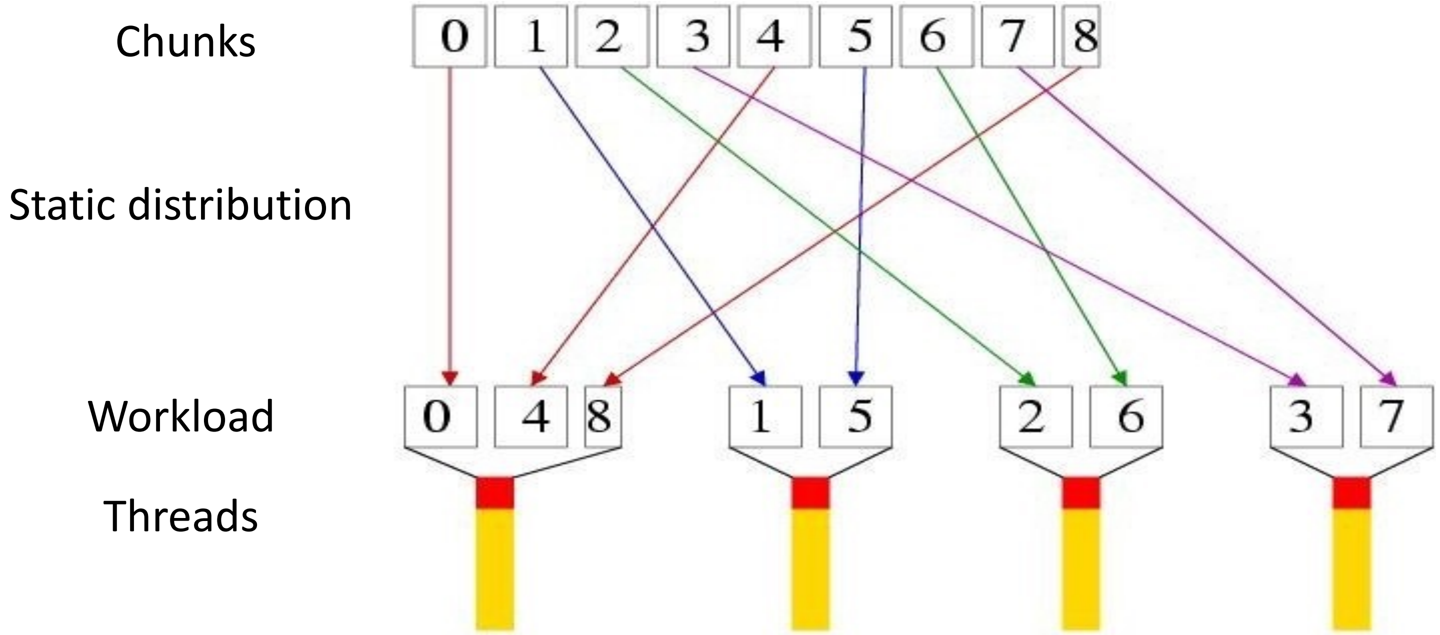
- Choosing a right scheduling for a loop is a major criterion for a good load balancing.

- There is an implicit barrier at the end of a ***for*** construct unless a ***NOWAIT*** clause is specified.

Work Sharing – FOR directive – SCHEDULE clause

- ***Schedule(static,** chunk_size)* – Iterations are divided into chunks of *chunk_size.* The chunks are assigned to the threads of the team in a round-robin way in the order of the thread number.

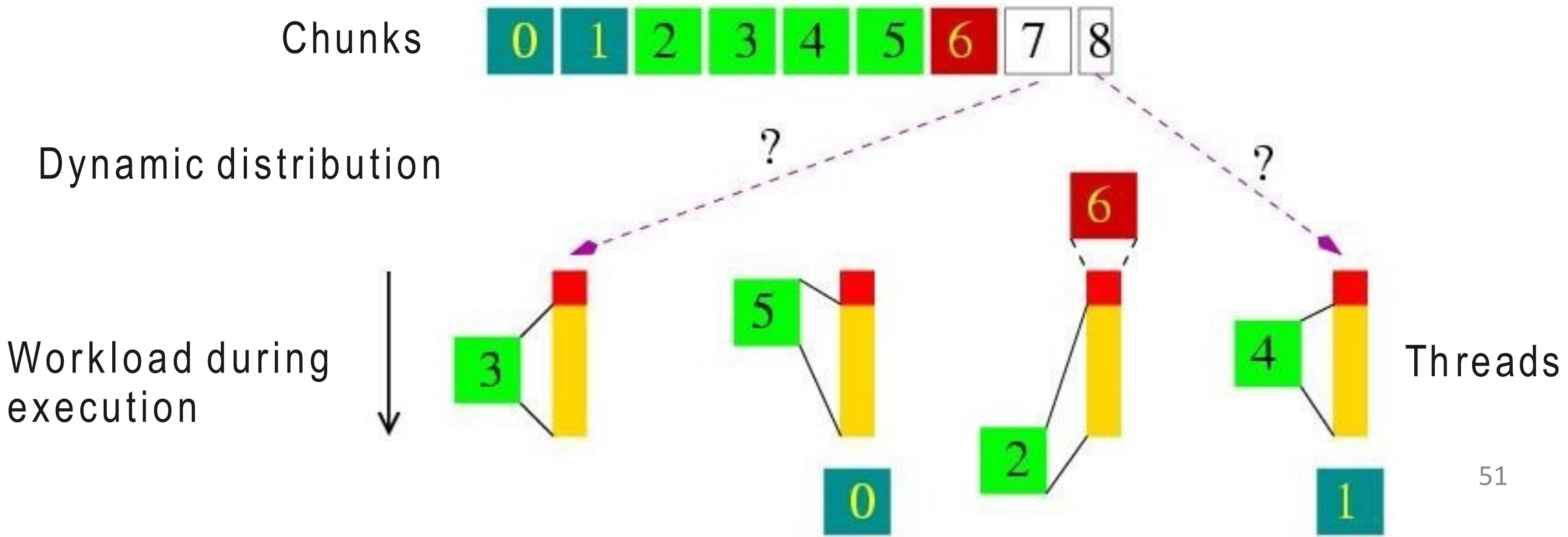
- If no *chunk_size* is specified the iteration space is divided into chunks that have almost the same size. At most only one chunk is assigned to one thread.



Work Sharing – FOR directive – SCHEDULE clause

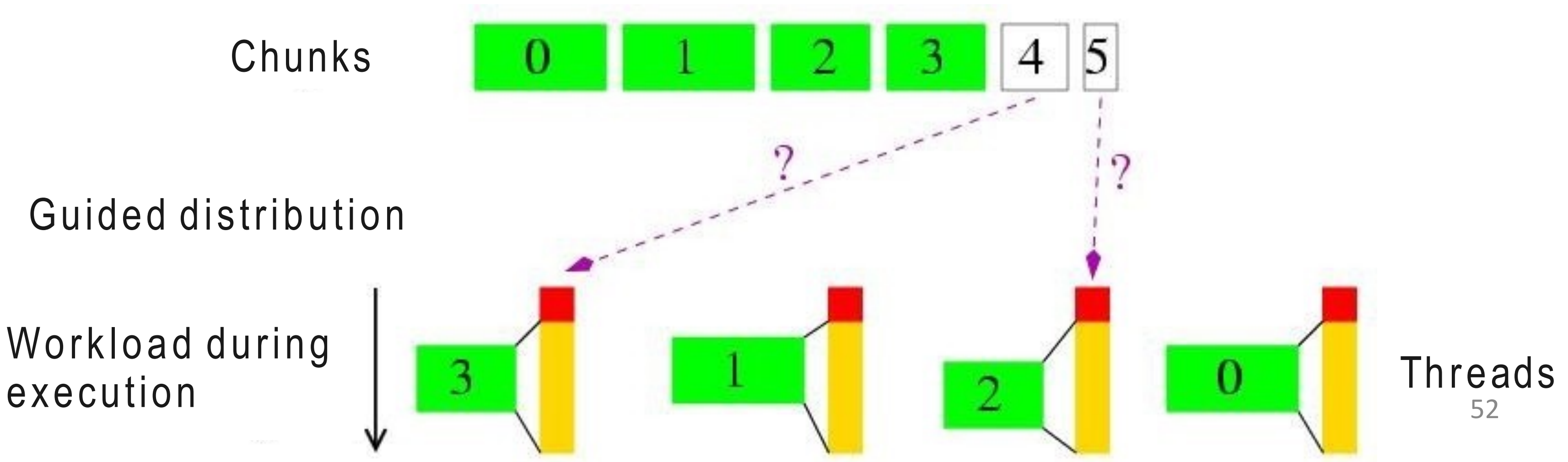
- ***Schedule(dynamic,** chunk_size)* – Iterations are distributed to the threads in chunks, of *chunk_size*, when the threads request them. Each thread executes a chunk, and it finishes, it requests another chunk until no more chunks to be distributed.

- If no *chunk_size* is specified the default is 1.



Work Sharing – FOR directive – SCHEDULE clause

- ***Schedule(guided,*** *chunk_size)* – Iterations are assigned to the threads in chunks when the threads request them. The size of each chunk is proportional to the number of unassigned iterations divided by the number of threads decreasing to *chunk_size.*
- If no *chunk_size* is specified the default is 1.



Question4

- static
- dynamic
- guided

What is the difference between these strategies?

Outline

- Parallel Architectures
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax
  - Parallel Region
  - Work Sharing
  - Exclusive Execution
  - Synchronization
  - Nested Parallelism
  - API + Env. Variables
  - Common mistakes
  - More Slides

Exclusive Execution

- In case the programmer wants to execute a task by only one thread and keep the other threads away from this task, two directives can be used:
- SINGLE
- MASTER

Exclusive Execution – SINGLE directive

- The ***SINGLE*** construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread).
- It is in general the first thread arriving to the ***SINGLE*** construct
- The other threads that do not execute the block wait at an implicit barrier at the end of the single construct unless a ***NOWAIT*** clause is specified.
- The different clauses that can be used in a ***SINGLE*** construct are:
  - ✓ Private
  - ✓ Firstprivate
  - ✓ Copyprivate
- The ***COPYPRIVATE*** clause uses a private variable to broadcast a value from one thread to the other threads of the team.

Exclusive Execution – SINGLE directive

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

int main(void)
{
  int rank;
  int a;

  #pragma omp parallel private(a,rank)
  {
    a = 50000;

     #pragma omp single
     {
         a = -50000;
     }

    rank=omp_get_thread_num();
    printf("Rank : %d ; A = %d\n",
               rank,a);
  }
  return 0;
}
```

```
> gcc …fopenmp o omp omp.c
> export OMP_NUM_THREADS=8
> ./omp

 Rank : 6 ; A = 50000
 Rank : 0 ; A = 50000
 Rank : 3 ; A = 50000
 Rank : 4 ; A = 50000
 Rank : 1 ; A = 50000
 Rank : 5 ; A = 50000
 Rank : 7 ; A = -50000
 Rank : 2 ; A = 50000
```

Exclusive Execution – SINGLE directive

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

int main(void)
{
  int rank;
  int a;

  #pragma omp parallel private(a,rank)
  {
    a = 50000;

   #pragma omp single copyprivate(a)
    {
        a = -50000;
    }

    rank=omp_get_thread_num();
    printf("Rank : %d ; A = %d\n",
               rank,a);
  }
  return 0;
}
```

```
> gcc …fopenmp o omp omp.c
> export OMP_NUM_THREADS=8
> ./omp

 Rank : 6 ; A = -50000
 Rank : 0 ; A = -50000
 Rank : 3 ; A = -50000
 Rank : 4 ; A = -50000
 Rank : 1 ; A = -50000
 Rank : 5 ; A = -50000
 Rank : 7 ; A = -50000
 Rank : 2 ; A = -50000
```

MASTER directive

- The area created by the MASTER directive is executed by the master thread (thread 0).
- No implicit synchronization at the end of the MASTER region.

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

int main()
{
    int rank;
    int a;

    #pragma omp parallel
  private(a,rank)
    {
        a = 50000;

        #pragma omp master
        {
            a = -50000;
        }

        rank=omp_get_thread_num();
        printf("Rank : %d ; A =
  %d\n",rank,a);
    }
    return 0;
}
```

```
> gcc …  fopenmp o omp omp.c
> export OMP_NUM_THREADS=8
> ./omp

 Rank : 6 ; A = 50000
 Rank : 0 ; A = -50000
 Rank : 3 ; A = 50000
 Rank : 4 ; A = 50000
 Rank : 1 ; A = 50000
 Rank : 5 ; A = 50000
 Rank : 7 ; A = 50000
 Rank : 2 ; A = 50000
```

Question5

- Why is Exclusive Execution needed?

Outline

- Parallel Architectures
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax
  - Parallel Region
  - Work Sharing
  - Exclusive Execution
  - Synchronization
  - Nested Parallelism
  - API + Env. Variables
  - Common mistakes
  - More Slides

Synchronization

- Synchronization is important:
  - ✓ To check that all threads executed the same number of instructions in the program.
  - ✓ To order the execution of different threads that are executing the same portion of code and impacting one or multiple shared variables that must be a coherent value in the memory.
  - ✓ To synchronize some threads (at least two) from the same team (lock).

Synchronization – Explicit BARRIER

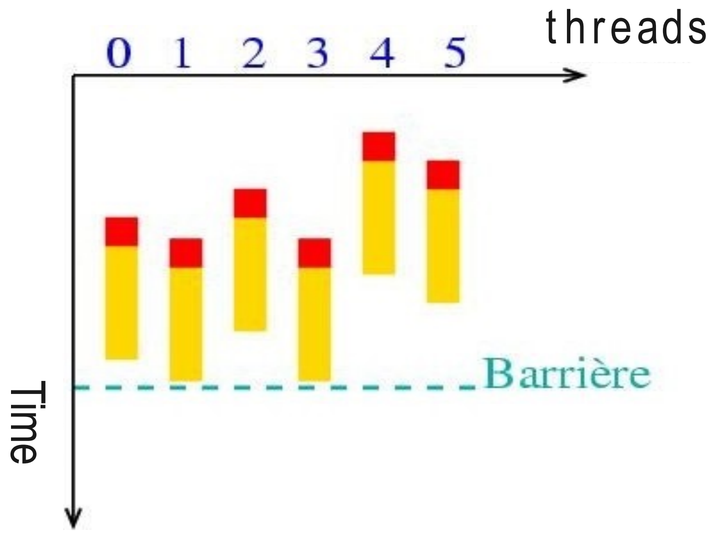
- There is an implicit barrier at the end of each ***PARALLEL*** construct at the end each ***FOR*** construct.

✓ This barrier can be bypassed if the clause ***NOWAIT*** is used.

- An explicit barrier can be forced for synchronization using the ***BARRIER*** directive.

- The ***BARRIER*** directive synchronizes all threads of the same team in a parallel region.

- Each thread waiting at a barrier do not continue the execution of the program until all threads reach the same barrier.



Synchronization – Explicit BARRIER

```
#pragma omp parallel private(TID)
{
    double start,stop;

    TID=omp_get_thread_num();

    start = omp_get_wtime();

    if (TID < omp_get_num_threads()/2)
        sleep(3);

    stop = omp_get_wtime();

    printf("%.0f seconds elapsed before barrier, thread %d\n",
                stop-start,TID);

    #pragma omp barrier

    stop = omp_get_wtime();

    printf("%.0f seconds elapsed after barrier, thread %d\n", stop
                -start,TID);
}
```

Synchronization – Explicit BARRIER

```
> gcc …  fopenmp o omp omp.c
> export OMP_NUM_THREADS=8
> ./omp

 0 seconds elapsed before barrier, thread 7
 0 seconds elapsed before barrier, thread 5
 0 seconds elapsed before barrier, thread 4
 0 seconds elapsed before barrier, thread 6
 3 seconds elapsed before barrier, thread 0
 3 seconds elapsed before barrier, thread 3
 3 seconds elapsed before barrier, thread 1
 3 seconds elapsed before barrier, thread 2
 3 seconds elapsed after barrier, thread 1
 3 seconds elapsed after barrier, thread 0
 3 seconds elapsed after barrier, thread 7
 3 seconds elapsed after barrier, thread 2
 3 seconds elapsed after barrier, thread 4
 3 seconds elapsed after barrier, thread 3
 3 seconds elapsed after barrier, thread 6
 3 seconds elapsed after barrier, thread 5
```

Synchronization – ATOMIC directive

- An ***ATOMIC*** directive ensures that a specific shared storage location is updated in memory by one thread at a time.
- Simultaneous read and writing in the same statement may result in a indeterminate value.
- The ***ATOMIC*** directive is applied on the instruction that immediately follows the construct.

Synchronization – ATOMIC directive

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

int main(void)
{
  int counter, rank;

  counter = 50000;

  #pragma omp parallel default(none) \
          private(rank) shared(counter)
  {
    rank=omp_get_thread_num();

    #pragma omp atomic
    counter++;

    printf("Rank : %d ; counter = %d\n",
                 rank,counter);
  }

  printf("In total, counter = %d\n",counter);
  return 0;
}
```

```
> gcc …  fopenmp o omp omp.c
> export OMP_NUM_THREADS=8
> ./omp

 Rank : 0 ; counter = 50002
 Rank : 7 ; counter = 50001
 Rank : 1 ; counter = 50003
 Rank : 3 ; counter = 50004
 Rank : 4 ; counter = 50005
 Rank : 2 ; counter = 50006
 Rank : 6 ; counter = 50007
 Rank : 5 ; counter = 50008
 In total, counter = 50008
```

Synchronization – CRITICAL directive

- A ***CRITICAL*** directive ensures that a specific region in the program is executed by one thread at a time.

- A ***CRITICAL*** directive can be considered as a general version of the ***ATOMIC*** directive.

- For better performance it is not recommended to use a ***CRITICAL*** directive to do an ***ATOMIC*** operation.

Question6

- What is Explicit BARRIER?
- Why are atomic operations needed?

Question2

Synchronization – CRITICAL directive

```
#include <stdio.h>

#include<omp.h>
int main(void)
{
    int s, p;

    s = 0, p = 1;

    #pragma omp parallel
    {
        #pragma omp critical
        {
            s++;
            p*=2;
        }
    }
    printf("Total sum and product: %d, %d\n",s,p);

    return 0;
}
```

Export OMP_NUM_THREADS=8

S=?

P=?

Outline

- Parallel Architectures
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax
  - Parallel Region
  - Work Sharing
  - Exclusive Execution
  - Synchronization
  - Nested Parallelism
  - API + Env. Variables
  - Common mistakes
  - More Slides

Nested Parallelism

- The OpenMP standard allows nested parallelism.
- This nesting consists in having a parallel region inside another parallel region.
- ATTENTION! The threads IDs are *local* to each parallel region. Different threads with the same IDs may exist!
- **Pros** – Exploit the parallelization at different levels.
- **Cons** – Overhead of the parallel region creation/destruction.

Nested Parallelism

```
#pragma omp parallel
{
   #pragma omp for
   for(i=0; i<n ; ++i) {
     ...
   }

   #pragma omp parallel
   {
     work(...);
   }
}
```

```
void work(...) {
   /* declarations */
   #pragma omp for
   for (j=0; j<m; ++j)
   {
     ...
   }
}
```

Outline

- Parallel Architectures
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax
  - Parallel Region
  - Work Sharing
  - Exclusive Execution
  - Synchronization
  - Nested Parallelism
  - API + Env. Variables
  - Common mistakes
  - More Slides

Programing Interface

- ***omp_set_num_threads()***
  - ✓ Defines the number of threads in a parallel region (unless a *num_threads* clause is specified).

- ***omp_get_num_threads()***
  - ✓ Returns the number of threads available in the current parallel region.

- ***omp_get_thread_num()***
  - ✓ Returns the ID of the current thread in the current team.

- ***omp_get_max_threads()***
  - ✓ Returns the maximum number of threads that can be created in one parallel region (unless the *num_thread* clause is specified).

Programing Interface

- ***omp_get_num_procs()***
  - ✔ Returns the number of available logical processors.
- ***omp_in_parallel()***
  - ✔ Indicates if we are in a parallel region or not.
- ***omp_set_nested()***
  - ✔ Activates parallel region nesting
- ***omp_get_nested()***
  - Indicates if the nesting is allowed or not.

Environment Variables

- ***OMP_NUM_THREADS***
  - ✓ Maximum number of threads for a parallel region

- ***OMP_SCHEDULE*** *= static|dymanic|guided [chunk_size]*
  - ✓ Strategy chosen when the ***schedule(runtime)*** clause is specified in the code.

- ***OMP_NESTED***
  - ✓ Indicates to the OpenMP runtime if the nesting parallelism is allowed.

Outline

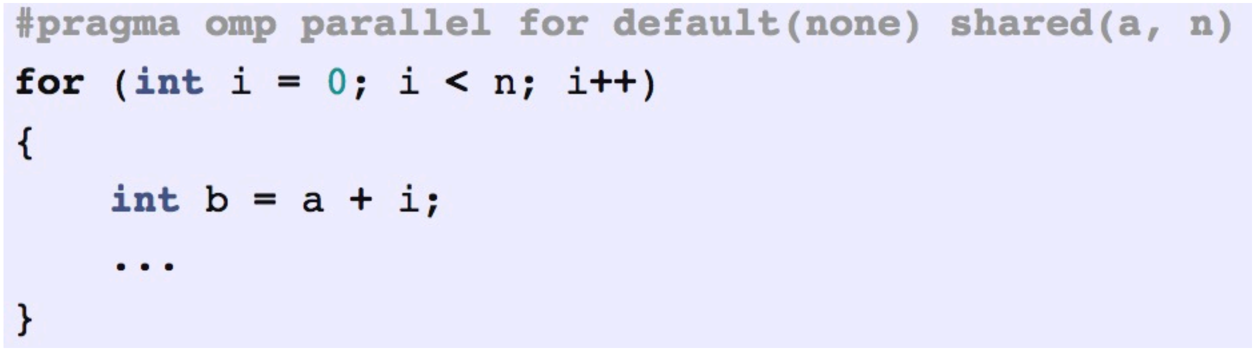
- Parallel Architectures
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax
  - Parallel Region
  - Work Sharing
  - Exclusive Execution
  - Synchronization
  - Nested Parallelism
  - API + Env. Variables
  - Common mistakes
  - More Slides

Common mistakes

Default Shared Attributes

- The *implicit* trap:
  - ✓ What is the relation between the variables?
  - ✓ Solution: be explicit on the variable status →

!!!
!!!!! USE DEFAULT(NONE) !!!!!
!!!


```
#pragma omp parallel for default(none) shared(a, n)
for (int i = 0; i < n; i++)
{
    int b = a + i;
    ...
}
```

Common mistakes

Default Shared Attributes

```
int main(int argc, char **argv)
{
     int i, n;
     int h, x, sum;

     n = atoi(argv[1]);
     h = 2;
     sum = 0;

     #pragma omp parallel for reduction(+:sum) shared(h)
      for (i=0 ; i<=n ; i++) {
        x = h * (i + 5);
        sum += (1 + x*x);
      }

     return h * sum;
 }
```

Common mistakes

Private Variables

- ✓ A private variable has an undefined value at the entry of a parallel region.
- ✓ The value of the original variable is undefined at the exit of the parallel region.

```
int n = 10;                  // shared
int a = 7;                   // shared

#pragma omp parallel for
for (int i = 0; i < n; i++) // i private
{
    int b = a + i;           // b private
    ...
}
```

Question3

Common mistakes

Private Variables

```
int main(int argc, char **argv)
{
    int i,a,b,c,n;

    n=atoi(argv[1]);
    a = b = 0;

    #pragma omp parallel for private(i,a,b)
    for (i=0; i<n; ++i) {
        ++b;
        a = b + i;
    }
    c = a + b;
    return c;
}
```

n=4
What is the final c?

Question4

Common mistakes

Private Variables

```
int main(int argc, char **argv)
{
    int i,a,b,c,n;

    n=atoi(argv[1]);
    b = 0;

    #pragma omp parallel for firstprivate(b)\
                             lastprivate(a,b)
    for (i=0; i<n; ++i) {
        ++b;
        a = b + i;
    }
    c = a + b;
    return c;
}
```

n=4

What is the final c?

Common mistakes

Bad Use of Master Construct

- The programmer may forget that there is ***no implicit barrier*** at the end of the ***MASTER*** construct.

```
int main(void)
{
    int xInit, xLocal;

    #pragma omp parallel shared(xInit)
    private(xLocal)
    {
        #pragma omp master
        { xInit = 10; }

        xLocal = xInit;
    }
}
```

Common mistakes

- Summary
  - Use ***default(none)***
  - Check the scope of variables in the parallel region.
  - Initialize the private variables.

Outline

- Parallel Architectures
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax
  - Parallel Region
  - Work Sharing
  - Exclusive Execution
  - Synchronization
  - Nested Parallelism
  - API + Env. Variables
  - Common mistakes
  - More Slides

SECTIONS directive

- A section is a portion of code or a block executed by one thread.
- The ***SECTIONS*** directive is a non-iterative construct that contains several blocks executed by different threads. Each block is executed by one thread.
- The different blocks are independent.
- Different blocks may be defined by the programmer using the ***SECTION*** directive inside a ***SECTIONS*** construct.

SECTIONS directive

- All ***SECTION*** directives should be defined inside the lexical scope of a ***SECTIONS*** directive.
- There is an implicit barrier at the end of the ***SECTIONS*** construct unless a ***NOWAIT*** clause is specified.
- The different clauses that can be used in a ***SECTIONS*** directive are:
  - ✓ Private
  - ✓ Firstprivate
  - ✓ Lastprivate
  - ✓ Reduction

SECTION directive

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

int main()
{
  int   rank;

  #pragma omp parallel private(rank)
  {
      rank=omp_get_thread_num();
      #pragma omp sections nowait
      {
       #pragma omp section
       {
            printf("[Thread %d] \n",
   rank);
       }
       #pragma omp section
       {
            printf("[Thread %d] \n",
   rank);
       }
      }
  }
  return 0;
}
```

```
> gcc …  -fopenmp o omp omp.c
> export OMP_NUM_THREADS=8
> ./omp

[Thread 7]
[Thread 0]
```

Summary

- Parallel Architectures
  - SMP： Symmetric Multi-Processor
  - NUMA： Non-Uniform Memory Access
- Programing Models
- OpenMP-Introduction
- OpenMP-Syntax
  - Parallel Region
  - Work Sharing
  - Exclusive Execution
  - Synchronization
  - Nested Parallelism
  - API + Env. Variables
  - Common mistakes
  - More Slides

Thank you！

中国人民大學

RENMIN UNIVERSITY OF CHINA