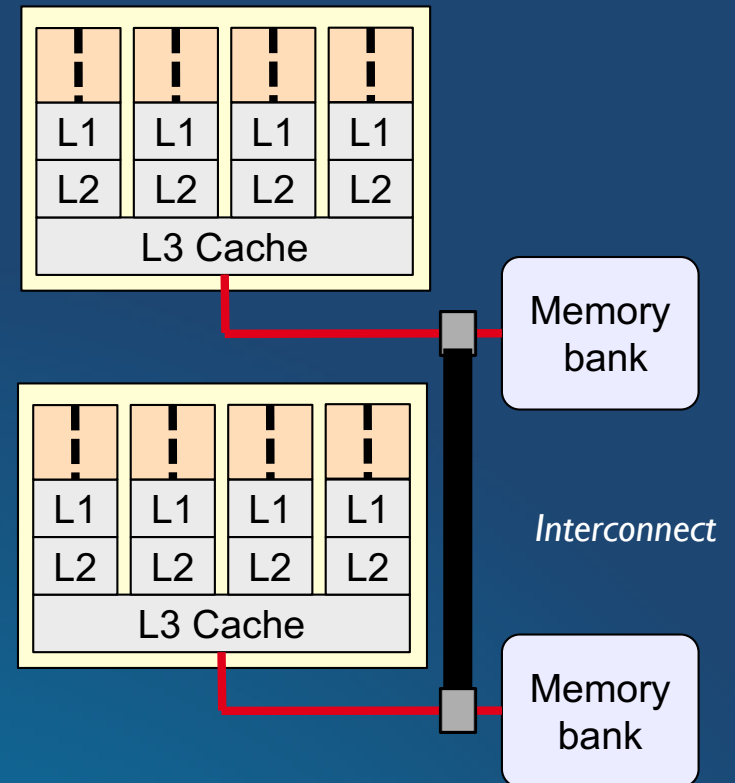


Shared Memory Machines and OpenMP Programming

Frédéric Desprez

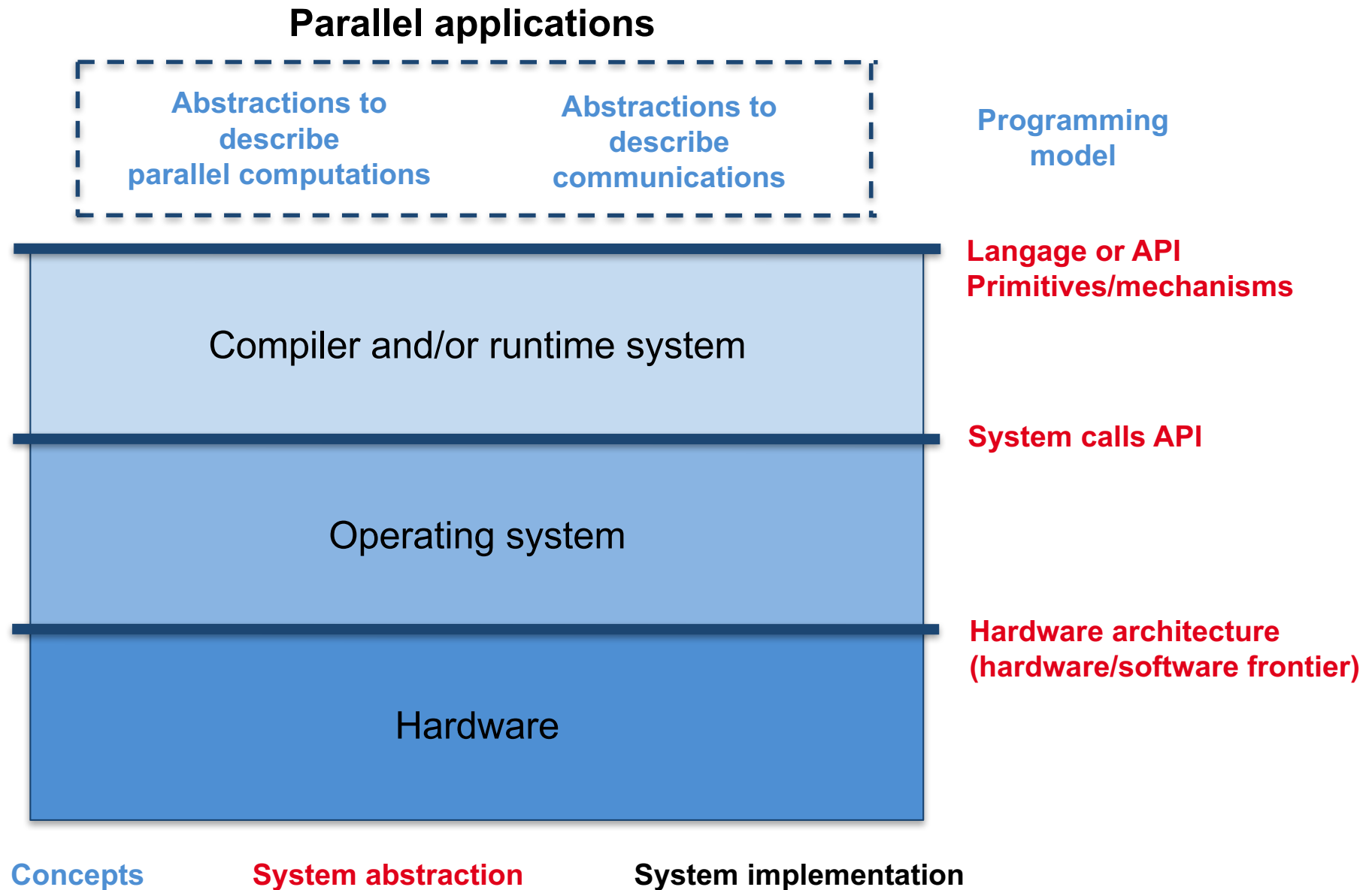
INRIA



Some references

- **OpenMP web site**
 - <http://www.openmp.org>
 - <http://www.openmp.org/specifications/>
- **OpenMP lecture**, François Broquedis (Corse), CERMACS School 2016
 - <http://smai.emath.fr/cemracs/cemracs16/programme.php>
- **OpenMP lecture**, Françoise Roch (Grenoble)
- **IDRIS lecture and lab work**
 - <http://www.idris.fr/formations/openmp/>
- **Using OpenMP , Portable Shared Memory Model**, Barbara Chapman
- **Parallel Programming in C with MPI and OpenMP**, M.J. Quinn
- **Programming Models for Parallel Computing**, P. Balaji
- **Parallel Programming – For Multicore and Cluster System**, T. Rauber, G. Rünger

Introduction



Introduction

Programming model: how to write (and describe) a parallel program

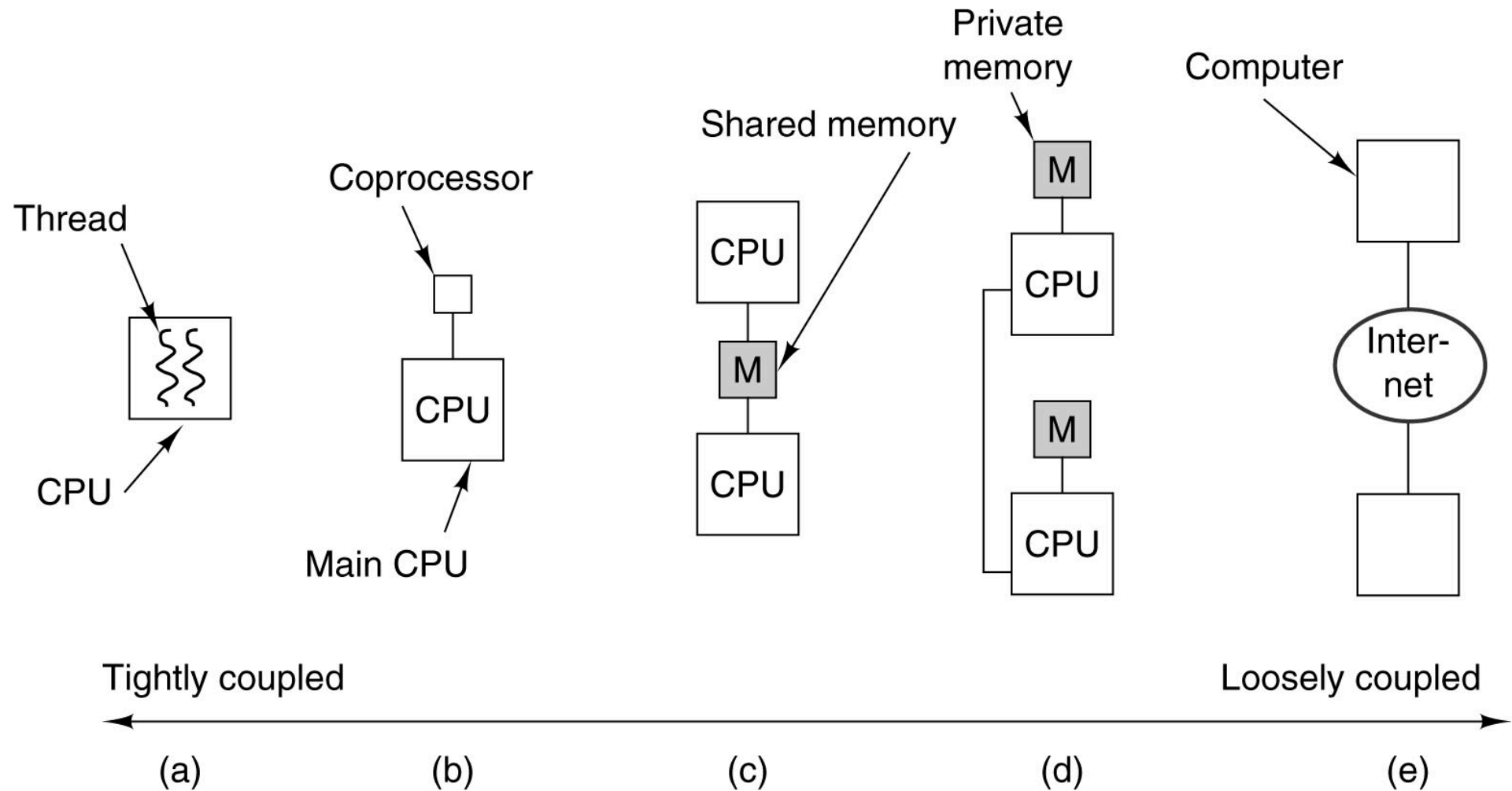
We will learn MPI (Message Passing Interface)

- The programmer manages everything (data distribution, computation distribution, processors synchronization, data exchanges)
- **Advantages**
 - Greater control from the programmer
 - Performances (if the code is well written!)
- **Drawbacks**
 - Parallelism assembly code
 - Performance portability
 - Less transparency

Other solution

- Give more work to the compiler and to the runtime system!

Parallel architectures



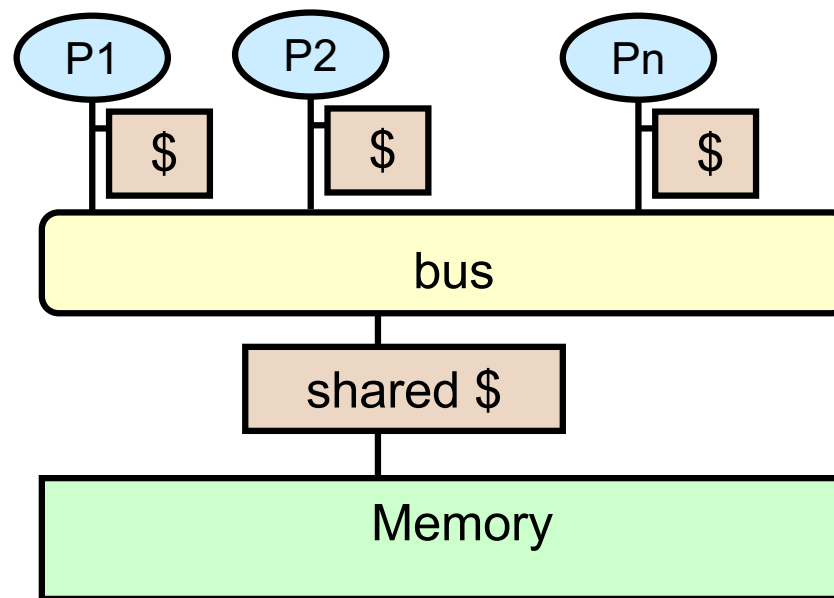
Shared memory machine model

Processors are connected to a large shared memory

- Also known as *Symmetric Multiprocessors* (SMPs)
- SGI, Sun, HP, Intel, SMPs IBM
- Multicore processors (except that caches are shared)

Scalability issues for large numbers of processors

- Usually ≤ 32 processors
- Uniform memory access (*Uniform Memory Access*, UMA)
- Lower cost for caches compared to the main memory



Note: \$ = cache

HPC architecture are getting more and more hierarchical

- **Parallelism is everywhere !**

- **At the architecture level**

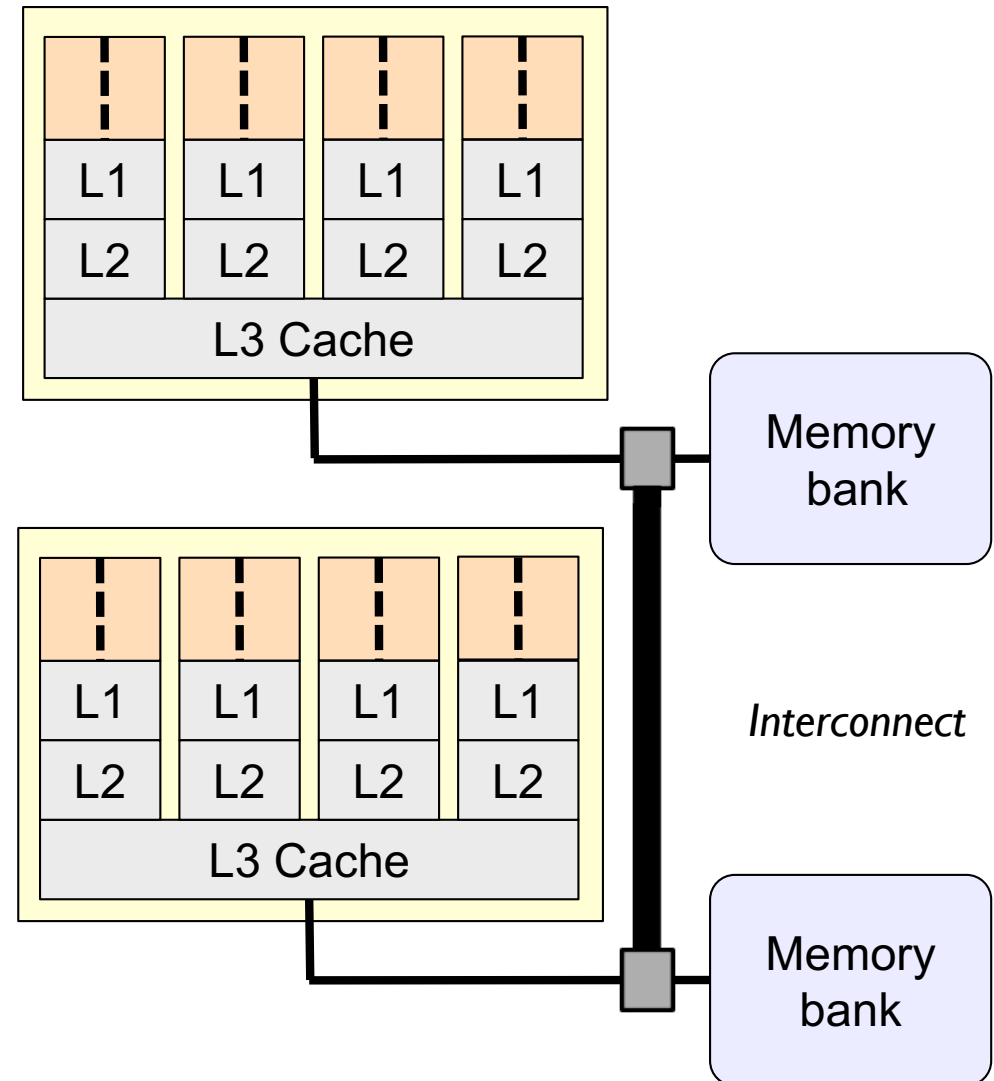
- SMP
- NUMA

- **At the processor level**

- Multicore chips

- **Current (solid) trend: back to the cc-NUMA era**

- AMD Hypertransport or Intel QuickPath to connect multicore chips together in a NUMA fashion



How to program these parallel machines?

- **The « good old » thread library**
 - A way to achieve the best performance for a particular instance of a problem (architecture, application, data set)
 - Not portable, most of the time...

The « user-friendly » (...) parallel programming environments

- MPI
 - Standard for distributed programming
- OpenMP
 - De-facto standard for shared-memory programming
- and all these great programming languages I won't talk about today
 - Cilk+, TBB, Charm++, UPC, X10, Chapel, OpenCL, OpenACC, ...



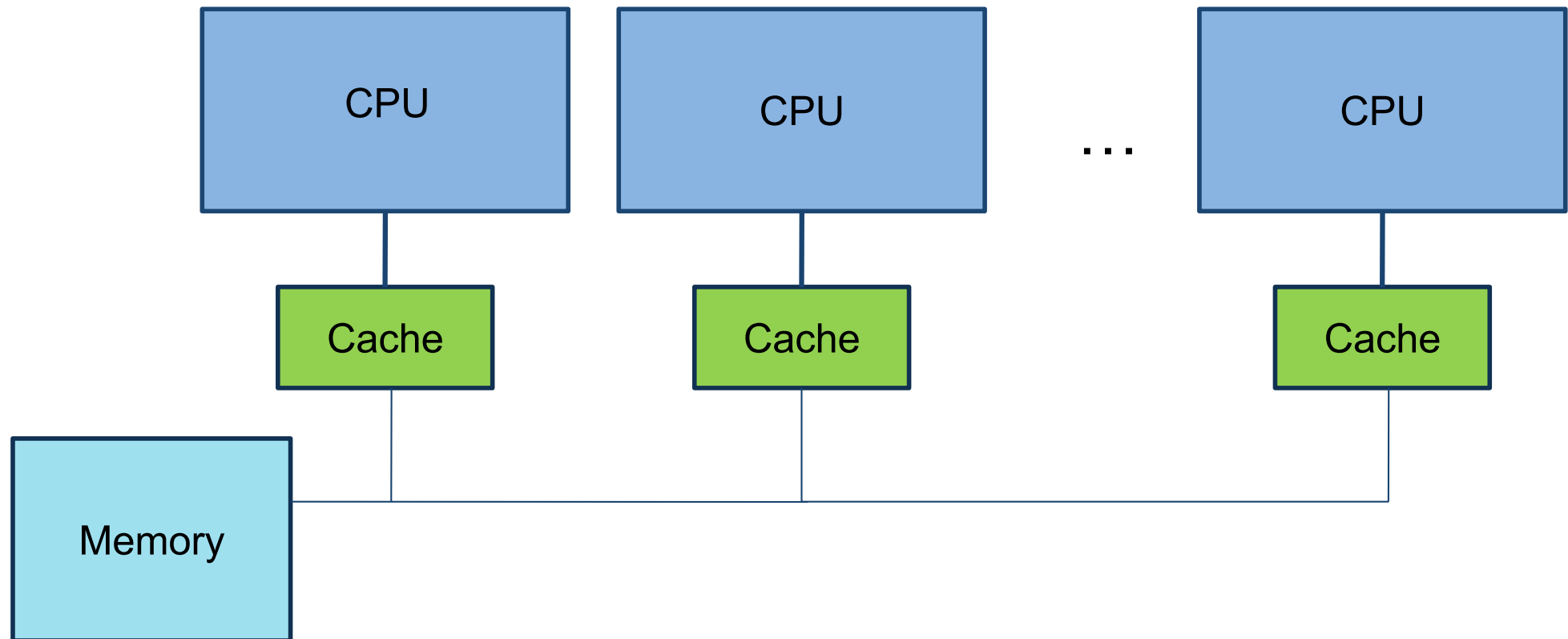
Multi-task programming model on shared memory architecture

- Several tasks are executed in parallel
- Memory is shared (physically or virtually)
- Communication between tasks is done by reads and writes in the shared memory
 - Eg. The general-purpose multi-core processors share a common memory
- Tasks can be assigned to distinct cores

Multi-task programming model on shared memory architecture

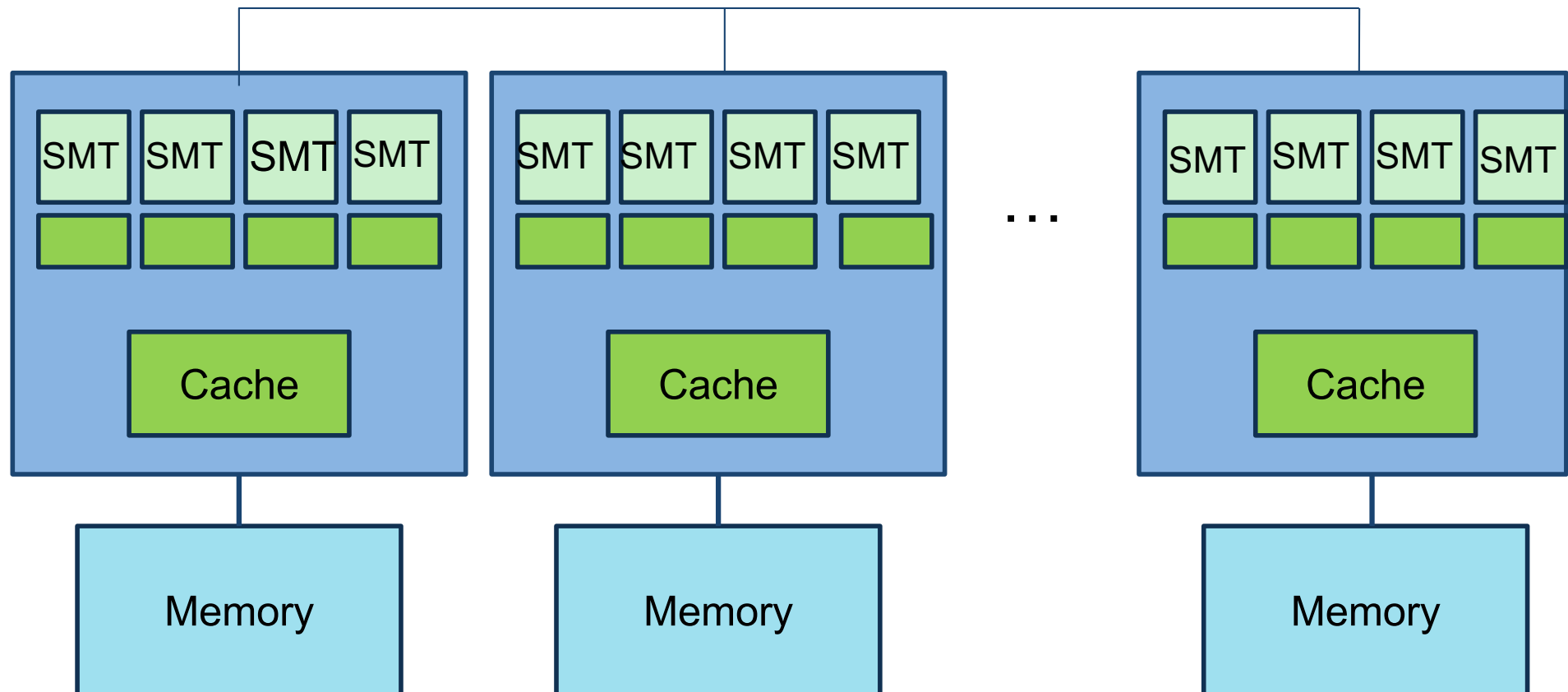
- The Pthreads library: POSIX thread library, adopted by most operating systems
 - The writing of a code requires a considerable number of lines specifically dedicated to threads
 - Example: parallelizing a loop involves
 - Declare thread structures,
 - create threads,
 - compute loop boundaries,
 - assign them to threads, ...
- OpenMP: a simpler alternative for the programmer

Multi-task programming on UMA architectures



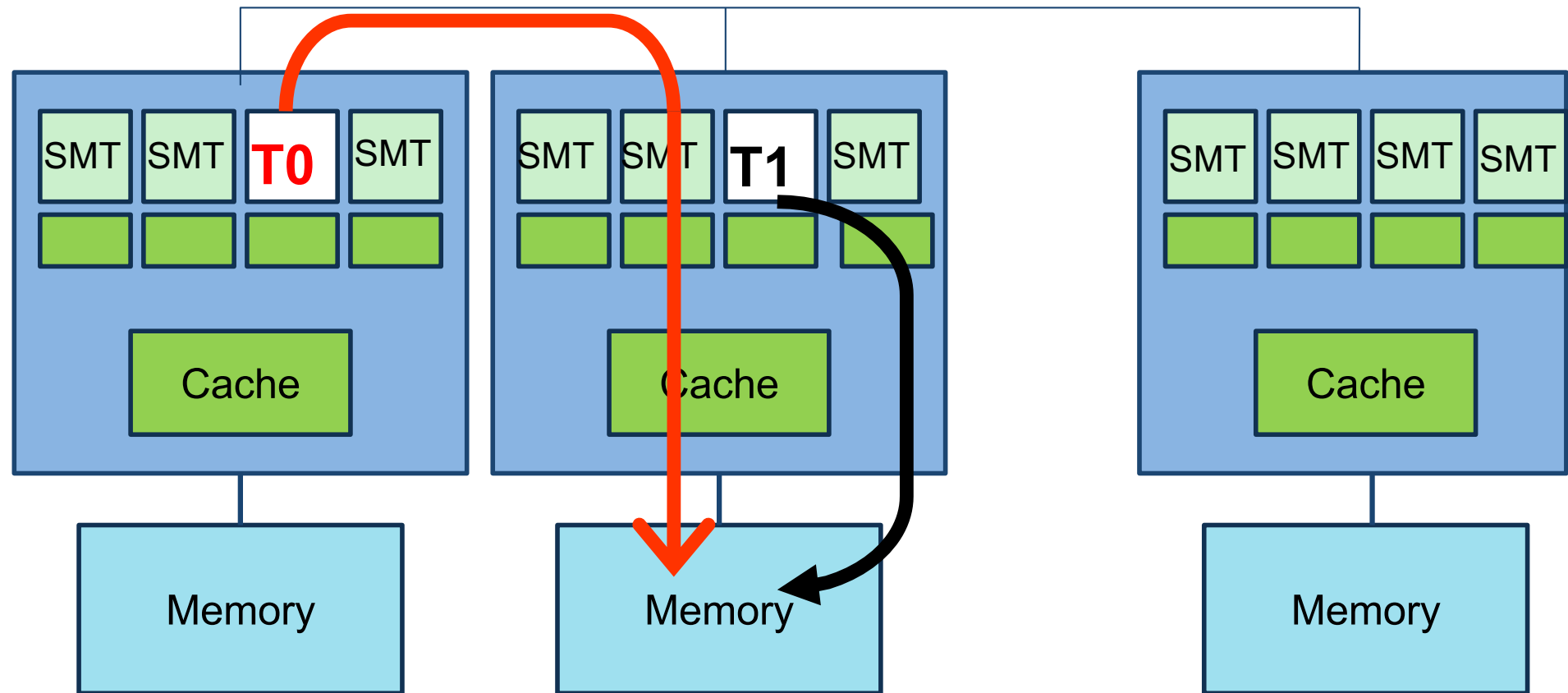
- Memory is shared
 - Uniform Memory Access Architectures (UMA)
 - An inherent problem: memory contentions

Multi-task programming on UMA multicore architectures



- Memory is directly attached to multicore chips
 - Non-Uniform Memory Access architectures (NUMA)

Multi-task programming on UMA multicore architectures



Distant access

Local access

OpenMP

- A de-facto standard API to write shared memory parallel applications in C, C++ and Fortran
- Consists of compiler directives, runtime routines and environment variables
- Specification maintained by the OpenMP Architecture Review Board (<http://www.openmp.org>)
- Current version of the specification: 4.5 (November 2015)
 - Next release 5.0

Advantages of OpenMP

- A **mature** standard
 - Speeding-up your applications since 1998
- **Portable**
 - Supported by many compilers, ported on many architectures
- Allows **incremental parallelization**
- Imposes low to **no overhead on the sequential execution** of the program
 - Just tell your compiler to ignore the OpenMP pragmas and you get back to your sequential program
- Supported by a wide and active community
 - The specifications have been moving fast since revision 3.0 (2008) to support:
 - new kinds of parallelism (tasking)
 - new kinds of architectures (accelerators)

OpenMP model characteristics

Advantages

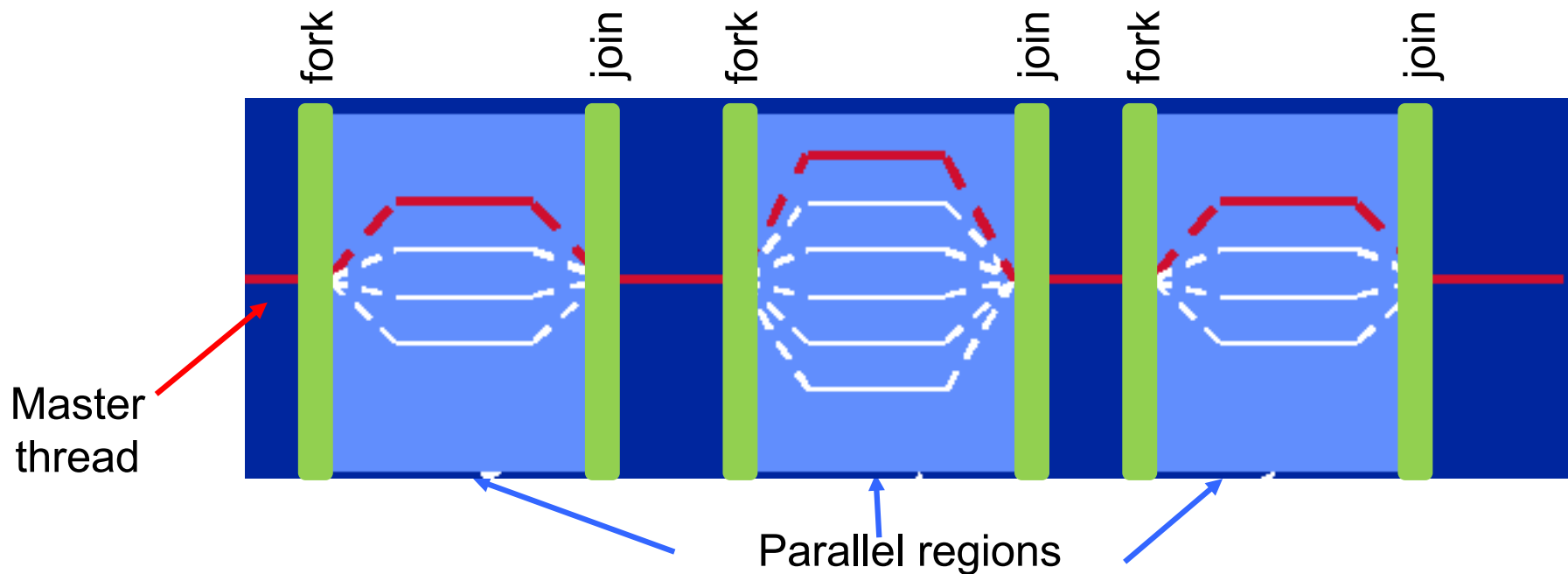
- Transparent and portable thread management
- Easy programming
- Can be used with MPI (hybrid parallelism)

Drawbacks

- Data locality problem
- Shared but non-hierarchical memory
- Efficiency not guaranteed (impact of the material organization of the machine)
- Limited scalability, moderate parallelism

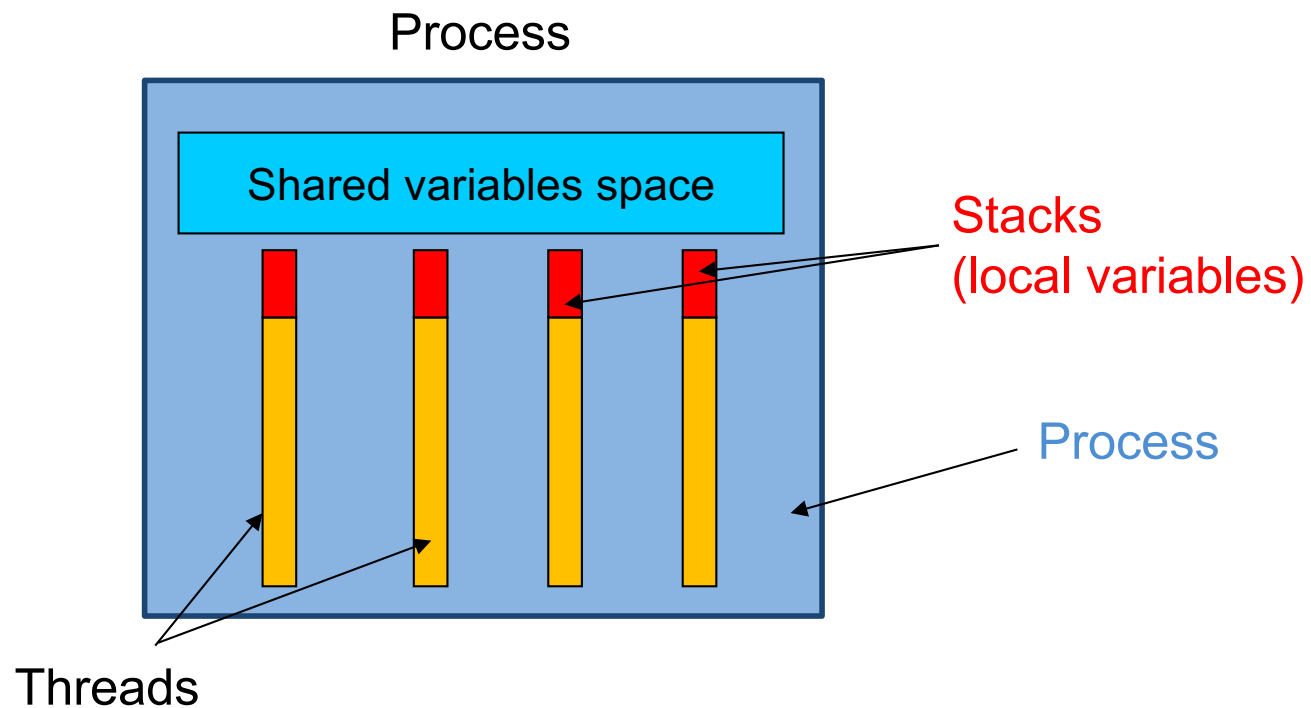
Introduction: execution model

- An OpenMP program is executed by a **unique process** (on one or many cores)
- **Fork-Join Parallelism**
 - Master thread spawns a team of threads as needed
 - Parallelism is **added incrementally**: that is, the sequential program evolves into a parallel program
 - Entering a parallel region will **create** some threads (*fork*)
 - Leaving a parallel region will **terminate** them (*join*)
 - Any statement executed outside parallel regions are executed **sequentially**



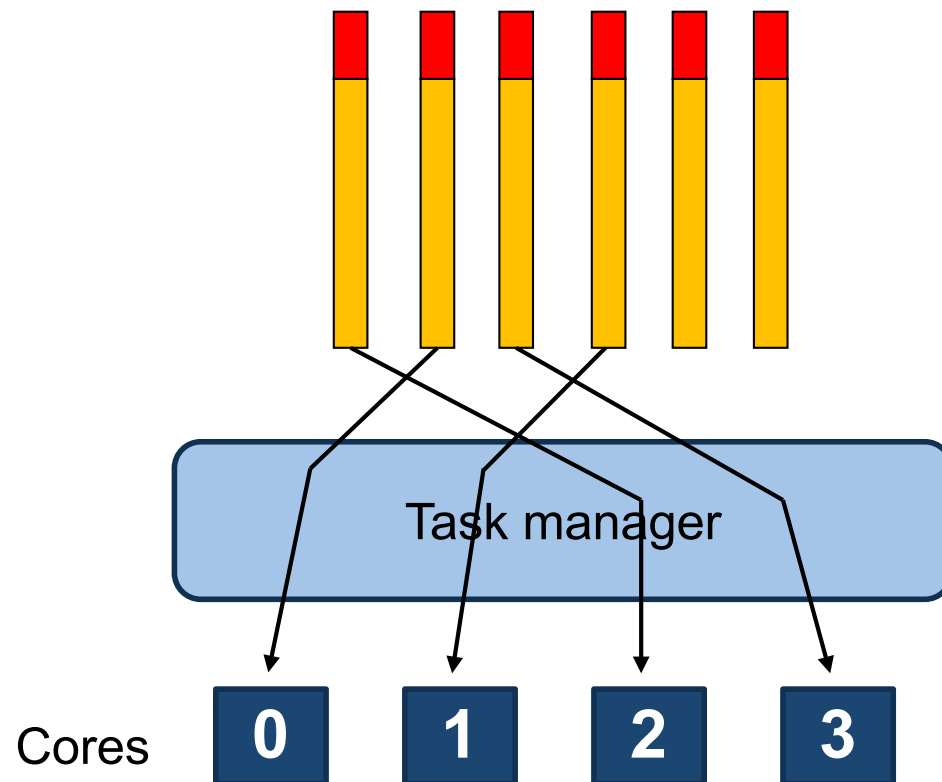
Introduction: threads

- Threads access the same resources as the main process
- They have a stack (stack, stack pointer and clean instructions pointer)

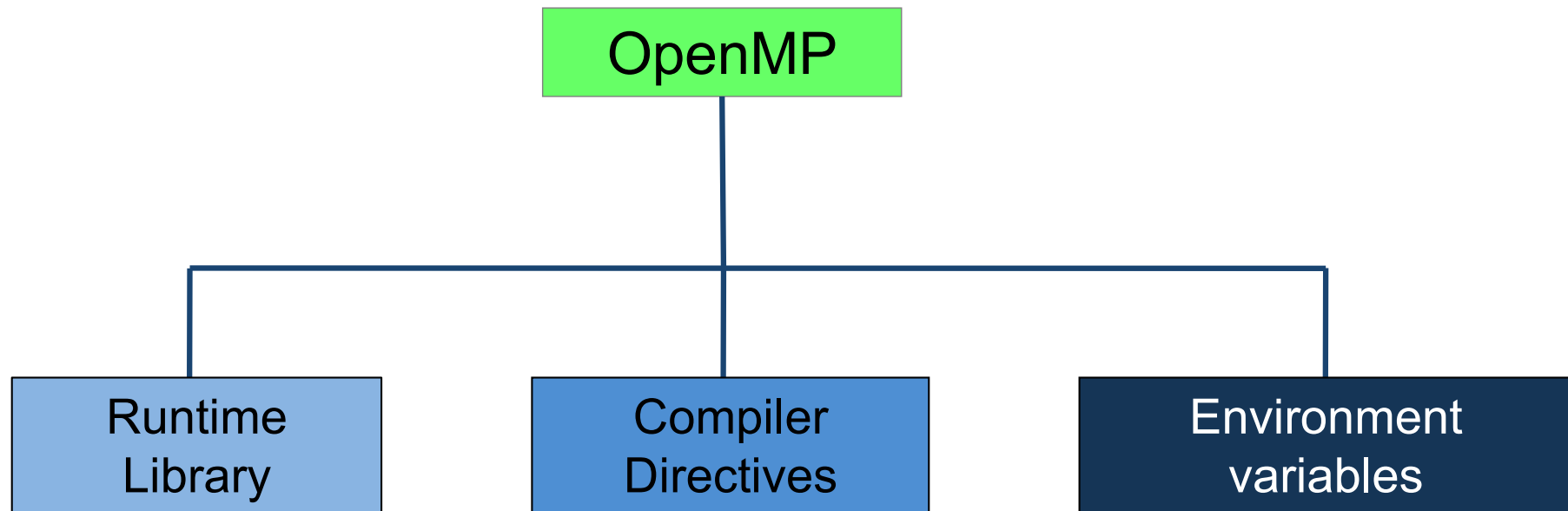


Introduction: execution of an OpenMP program on a multicore

The task management system of the operating system assigns the tasks on the cores



OpenMP structure: software architecture



OpenMP structure: directives/pragmas formats

- directive [clause[clause]..]

Fortran

```
!$OMP PARALLEL PRIVATE(a,b) &  
!$OMP FIRSTPRIVATE(c,d,e)  
...  
!$OMP END PARALLEL
```

C/C++

```
#pragma omp parallel private(a,b)  
                        firstprivate(c,d,e)  
{  
    ...  
}
```

- The line is interpreted if openmp option to the compiler call otherwise comment
→ portability

OpenMP structure: prototyping

We have

- A Fortran 95 module OMP_LIB
- An C/C++ input file omp.h

that define the prototypes of all the functions of the OpenMP library

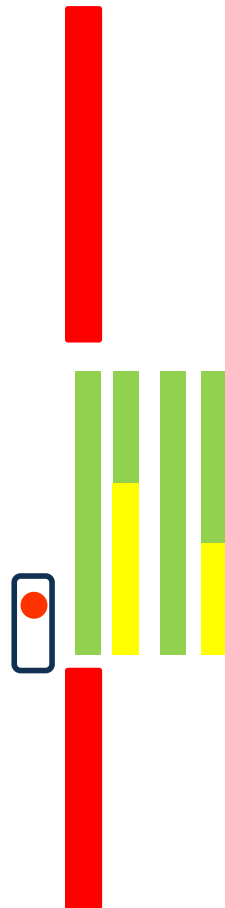
Fortran

```
Program example
!$ USE OMP_LIB
!$OMP PARALLEL PRIVATE(a,b) &
...
tmp= OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
```

C/C++

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

OpenMP structure: construction of a parallel region



fortran

PROGRAM example

!\$ USE OMP_LIB

Integer :: a, b, c

! Sequential code sequential executed by
the master

**!\$OMP PARALLELPRIVATE(a,b) &
!\$OMP SHARED(c)**

!
! Parallel zone executed by all the
! threads

!\$OMP END PARALLEL

! Sequential code

END PROGRAM example

C/C++

#include <omp.h>

main () {

Int a,b,c;

/* Sequential code sequential executed by
the master */

**#pragma omp parallel private(a,b) **
shared(c)

{
/* Parallel zone executed by all the
threads */
}

/* Sequential code */

}

Hello world !

```
void main()
{
    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
```

OpenMP's Hello world !

```
#include "omp.h"
void main()
{
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
}
```

OpenMP include file

Parallel region with default number of threads

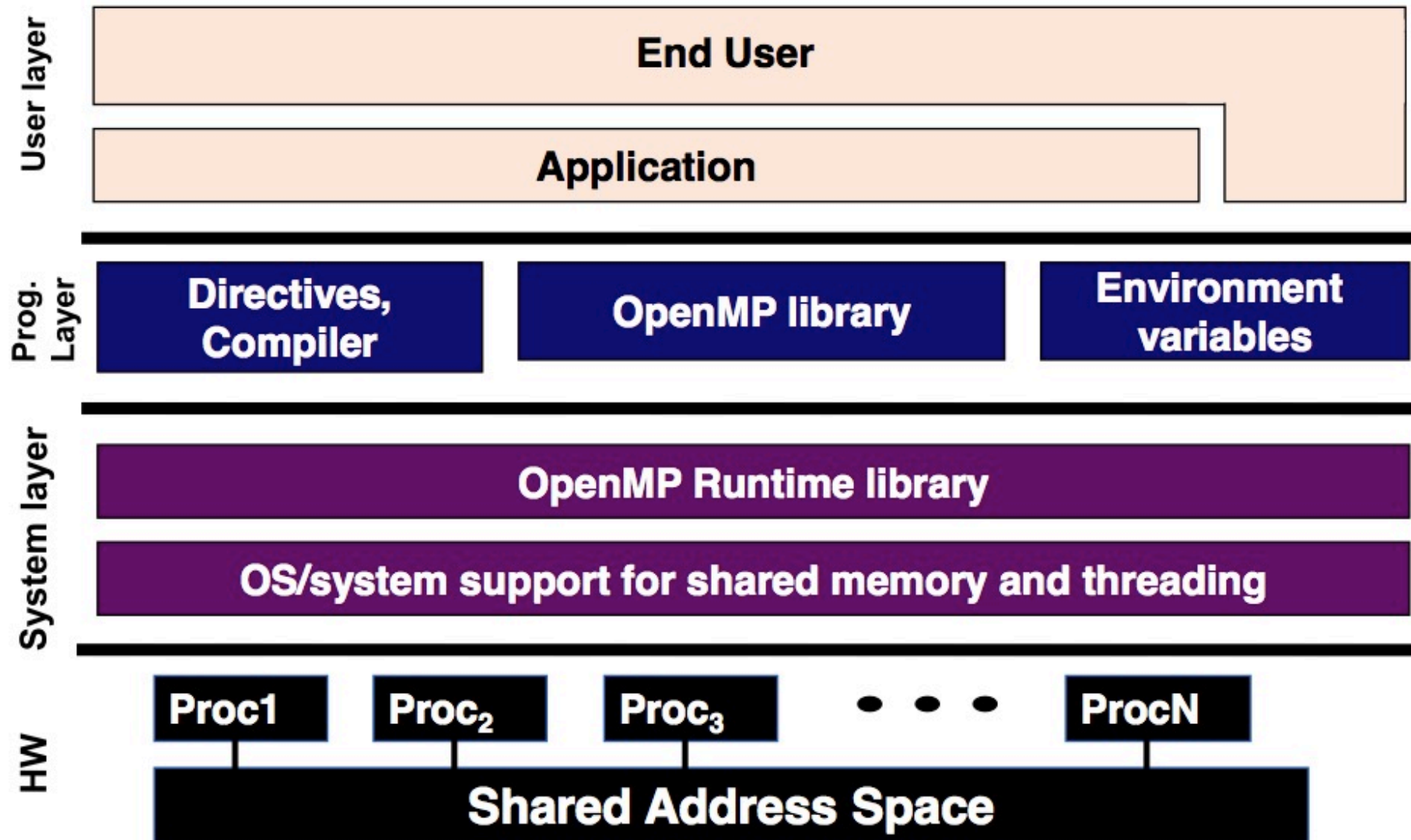
Runtime library function to return a thread ID.

End of the Parallel region

Sample Output

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

OpenMP Basic Defs: Solution Stack



IF clause of the PARALLEL directive

Conditional creation of a parallel region **IF(logical_expression)** clause

```
_____ fortran _____  
  
! Sequential code  
  
!$OMP PARALLEL IF(expr)  
  
! Parallel or sequential code depending of the expr value  
  
!$OMP END PARALLEL  
  
! Sequential code
```

The logical expression will be evaluated before entering the parallel region

How do threads interact?

- OpenMP is a multi-threading, shared address model
 - Threads communicate by sharing variables
- Unintended sharing of data causes race conditions:
 - race condition: when the program's outcome changes as the threads are scheduled differently
- To control race conditions
 - Use synchronization to protect data conflicts
- Synchronization is expensive so
 - Change how data is accessed to minimize the need for synchronization

OpenMP threads

Number of threads definition

- Through an environment variable: OMP_NUM_THREADS
- Through the routine: OMP_SET_NUM_THREADS()
- Through the clause NUM_THREADS() of the PARALLEL directive

Threads are numbered

- The number of threads is not necessary equal to the number of physical cores
- thread #0 is the master task
- OMP_GET_NUM_THREADS(): number of threads
- OMP_GET_THREAD_NUM(): thread number
- OMP_GET_MAX_THREADS(): maximum number of threads

OpenMP structure: compilation and execution

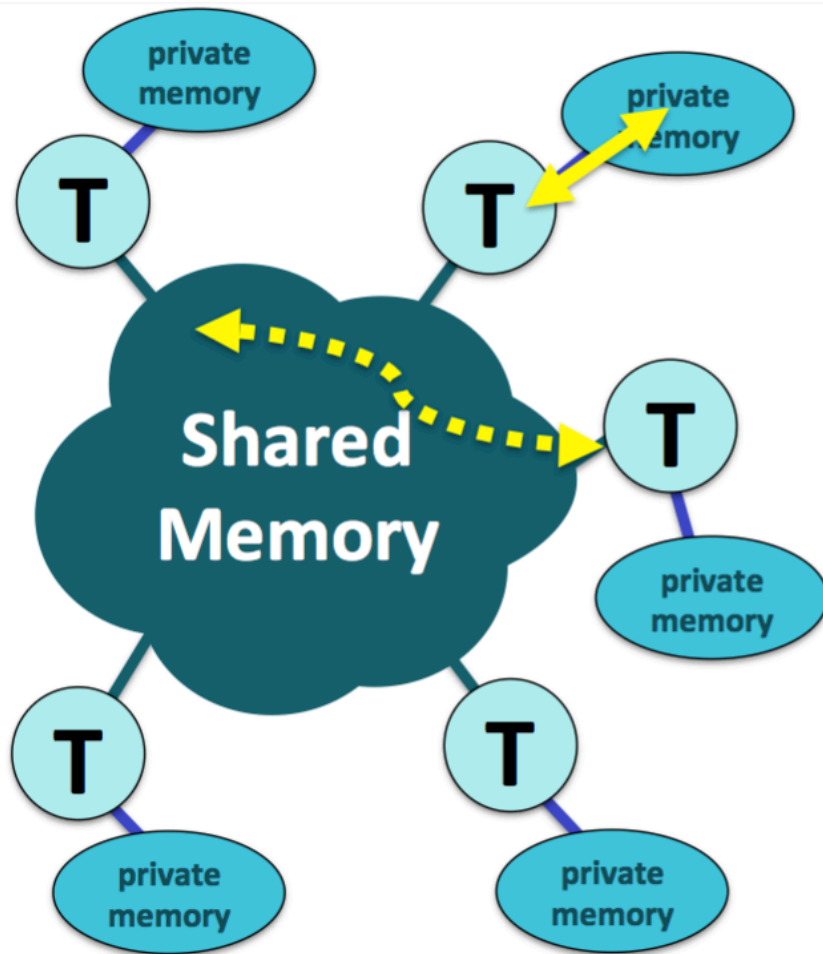
ifort (ou icc) -openmp prog.f	(INTEL)
f90 (ou cc ou CC) -openmp prog.f	(SUN Studio)
gcc/gfortran -fopenmp -std=f95 prog.f	(GNU)
export OMP_NUM_THREADS=2	

./a.out

ps -eLF

USER	PID	PPID	LWP	C	NLWP	SZ	RSS	PSR	...
------	-----	------	-----	---	------	----	-----	-----	-----

The OpenMP memory model



- All the threads have access to the same **globally shared** memory
- Each thread has access to **its own private memory area** that can not be accessed by other threads
- Data transfer is performed through shared memory and is **100% transparent to the application**
- The application programmer is responsible for providing the corresponding data-sharing attributes

Data sharing attributes

- Need to set the visibility of each variable that appears inside an OpenMP parallel region using the following **data-sharing attributes**
- **shared**: the data can be read and written by any thread of the team. All changes are visible to all threads
- **private**: each thread is working on its own version of the data that cannot be accessed by other threads of the team
- **firstprivate**: each thread is working on its own version of the variable. The data is initialized using the value it had before entering the parallel region
- **lastprivate**: each thread is working on its own version of the variable. The value of the last thread leaving the region is copied back to the variable.

Variable status

The status of a variable in a parallel zone

- SHARED, it's located in the global memory
- PRIVATE, it's located in the memory of each thread. It's value is undefined at the entrance of the zone
- Declaring the variable status
 - # pragma omp parallel private (list)
 - # pragma omp parallel firstprivate (list)
 - # pragma omp parallel shared (list)
- Declaring a default status
 - DEFAULT(PRIVATE|SHARED|NONE) clause

```
program private_var.f
  !$USE OMP_LIB
  integer:: tmp =999
  Call OMP_SET_NUM_THREADS(4)
  !$OMP PARALLEL PRIVATE(tmp)
    print *, tmp
    tmp = OMP_GET_THREAD_NUM()
    print *, OMP_GET_THREAD_NUM(), tmp
  !$OMP END PARALLEL
  print *, tmp
end
```

Putting Threads to Work: the Worksharing Constructs

```
1 void simple_loop(int N,  
2                 float *a,  
3                 float *b)  
4 {  
5     int i;  
6     // i, N, a and b are shared by  
7     // default  
8     #pragma omp parallel firstprivate(N)  
9     {  
10        // i is private by default  
11        #pragma omp for  
12        for (i = 1; i <= N; i++) {  
13            b[i] = (a[i] + a[i-1]) / 2.0;  
14        }  
15    }
```

- **omp for** : distribute the iterations of a loop over the threads of the parallel region.
- Here, assigns N/P iterations to each thread, P being the number of threads of the parallel region.
- **omp for** comes with an implicit **barrier synchronization** at the end of the loop one can remove with the **nowait** keyword.

Work sharing

- Distributing a loop between threads (// loop)
- Distribution of several sections of code between threads, one section of code per thread (// sections)
- Running a portion of code on a single thread
- Execution of several occurrences of the same function by different threads
- Execution by different threads of different work units, tasks

Work sharing: parallel loop

DO Directive in Fortran, **for** in C

Parallelism by distribution of iterations of a loop

- The way in which the iterations can be distributed can be specified in the **SCHEDULE** clause (coded in the program or by an environment variable)
- A global synchronization is performed at the end of construction **END DO** (unless **NOWAIT**)
- Possibility to have several **DO** constructions in a parallel region
- The loop indices are integers and private
- Infinite loops and **do while** are not parallelizable

DO and PARALLEL DO Directives

```
Program loop
implicit none
integer, parameter :: n=1024
integer          :: i, j
real, dimension(n, n) :: tab
!$OMP PARALLEL
...              ! Replicated code
!$OMP DO
  do j=1, n      ! Shared loop
    do i=1, n    ! Replicated loop
      tab(i, j) = i*j
    end do
  end do
!$OMP END DO
!$OMP END PARALLEL
end program loop
```

```
Program parallelloop
implicit none
integer, parameter :: n=1024
integer          :: i, j
real, dimension(n, n) :: tab
!$OMP PARALLEL DO
  do j=1 n      ! Shared loop
    do i=1, n    ! Replicated loop
      tab(i, j) = i*j
    end do
  end do
!$OMP END PARALLEL DO
end program parallelloop
```

PARALLEL DO is a fusion of 2 directives

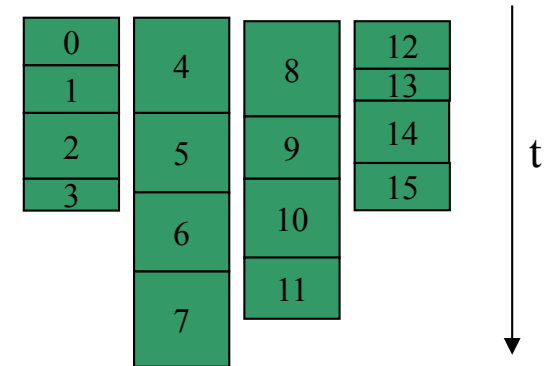
Beware: END PARALLEL DO includes a synchronization barrier!

Work sharing: SCHEDULE

!\$OMP DO SCHEDULE(STATIC, packet-size)

By default packet-size = $\#_iterations / \#_threads$

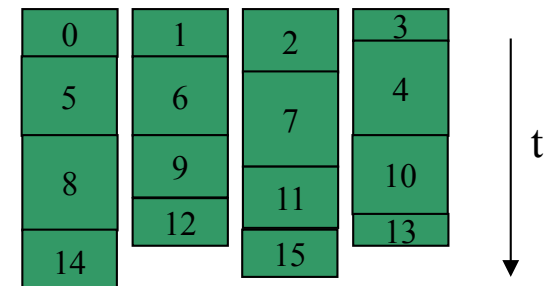
Ex: 16 iterations (0 to 15), 4 threads: packet size by default is 4



!\$OMP DO SCHEDULE(DYNAMIC, packet-size)

Packets are distributed to free threads in a dynamic way

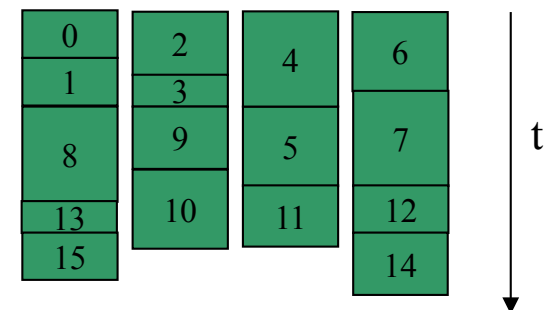
All the packets have the same size (except maybe the last one), by default the packet size is one



!\$OMP DO SCHEDULE(GUIDED, packet-size)

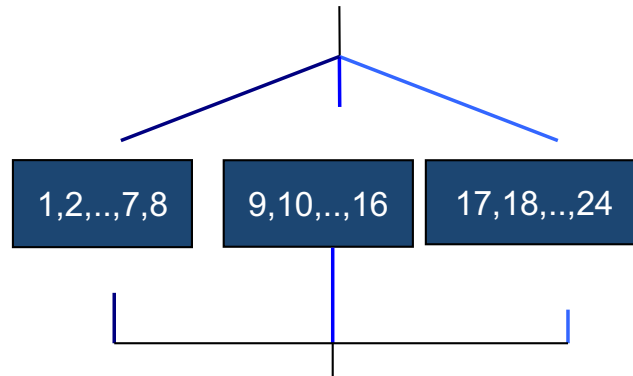
Packet-size: minimal packet size (1 by default) except the last one

Maximal packet size at the beginning of the loop (here 2) then decrease to balance the load

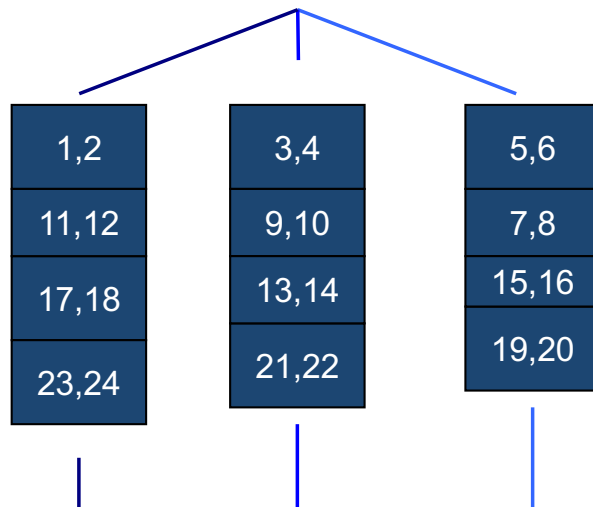


Work sharing: SCHEDULE

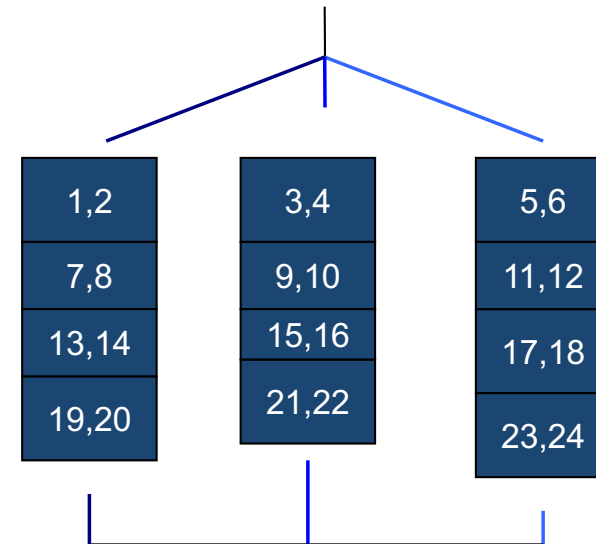
Ex: 24 iterations, 3 threads



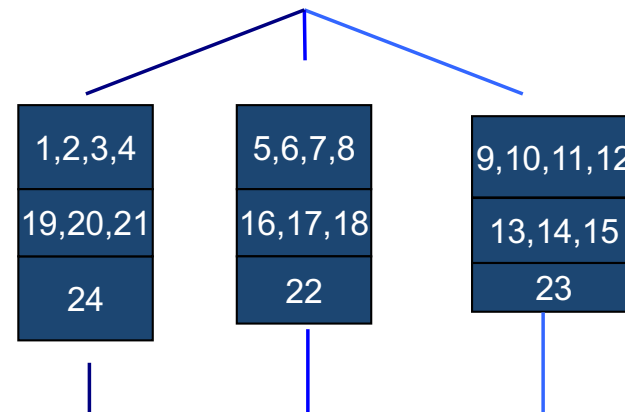
static mode with
Packet size = # iterations/# threads



Greedy: dynamic



Cyclic: static



Greedy: guided

Work sharing: SCHEDULE

The choice of the repartition mode can be delayed at the execution time using SCHEDULE(RUNTIME)

Taking into account the environment variable OMP_SCHEDULE

- Ex

```
export OMP_SCHEDULE="DYNAMIC,400"
```

A first example to illustrate OpenMP capabilities

Parallelize this simple code using OpenMP

```
f = 1.0
```

```
for (i = 0; i < N; i++)  
    z[i] = x[i] + y[i];
```

```
for (i = 0; i < M; i++)  
    a[i] = b[i] + c[i];
```

```
...
```

```
scale = sum (a, 0, m) + sum (z, 0, n) + f;
```

```
...
```

A first example to illustrate OpenMP capabilities

First create the parallel region and define the data-sharing attributes


```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
    f = 1.0

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];

    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];

    ...

    scale = sum (a, 0, m) + sum (z, 0, n) + f;
    ...
} /* End of OpenMP parallel region */
```

A blue L-shaped arrow is positioned to the right of the code. The vertical part of the arrow points upwards from the closing brace of the parallel region to the opening brace. The horizontal part of the arrow points to the left from the vertical part to the closing brace of the parallel region. The text "parallel region" is written vertically along the vertical part of the arrow.

A first example to illustrate OpenMP capabilities

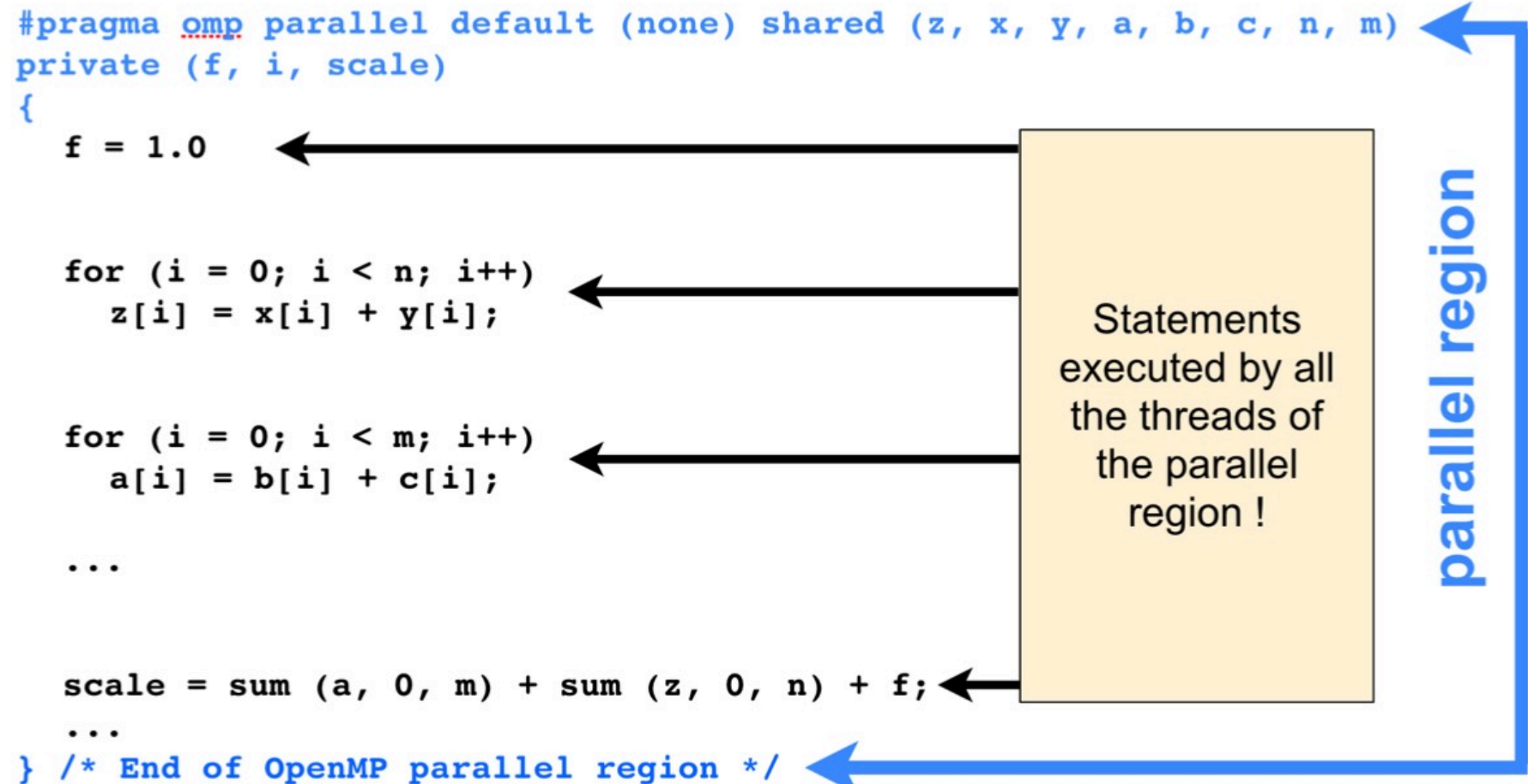
```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
    f = 1.0

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];

    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];

    ...

    scale = sum (a, 0, m) + sum (z, 0, n) + f;
    ...
} /* End of OpenMP parallel region */
```



At this point, all the threads execute the whole program (you won't get any speed-up from this!)

A first example to illustrate OpenMP capabilities

Now distribute the loop iterations over the threads using omp for

```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
    f = 1.0
    #pragma omp for
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
    #pragma omp for
    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];
    ...
    scale = sum (a, 0, m) + sum (z, 0, n) + f;
    ...
} /* End of OpenMP parallel region */
```

Statements executed by all the threads of the parallel region

parallel loop (work is distributed)

parallel loop (work is distributed)

parallel region

Statements executed by all the threads of the parallel region

Optimization #1: Remove Unnecessary Synchronizations

There are no dependencies between the two parallel loops, we remove the implicit barrier between the two

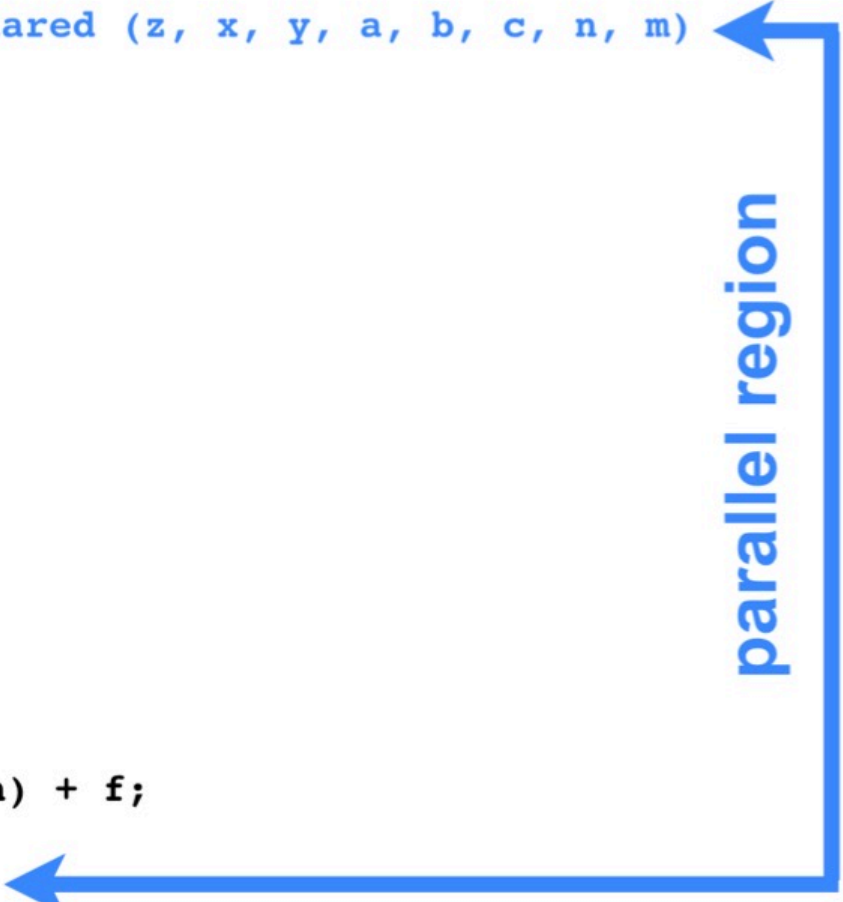
```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
    f = 1.0

    #pragma omp for nowait
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];

    #pragma omp for nowait
    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];

    ...

    #pragma omp barrier
    scale = sum (a, 0, m) + sum (z, 0, n) + f;
    ...
} /* End of OpenMP parallel region */
```



Optimization #2: Don't Go Parallel if the Workload is Small

We don't want to pay the price of thread management if the workload is too small to be computed in parallel

```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale) if (n > some_threshold && m > some_threshold)
{
    f = 1.0

    #pragma omp for nowait
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];

    #pragma omp for nowait
    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];

    ...

    #pragma omp barrier
    scale = sum (a, 0, m) + sum (z, 0, n) + f;
    ...
} /* End of OpenMP parallel region */
```