Shared Memory machines and OpenMP programming

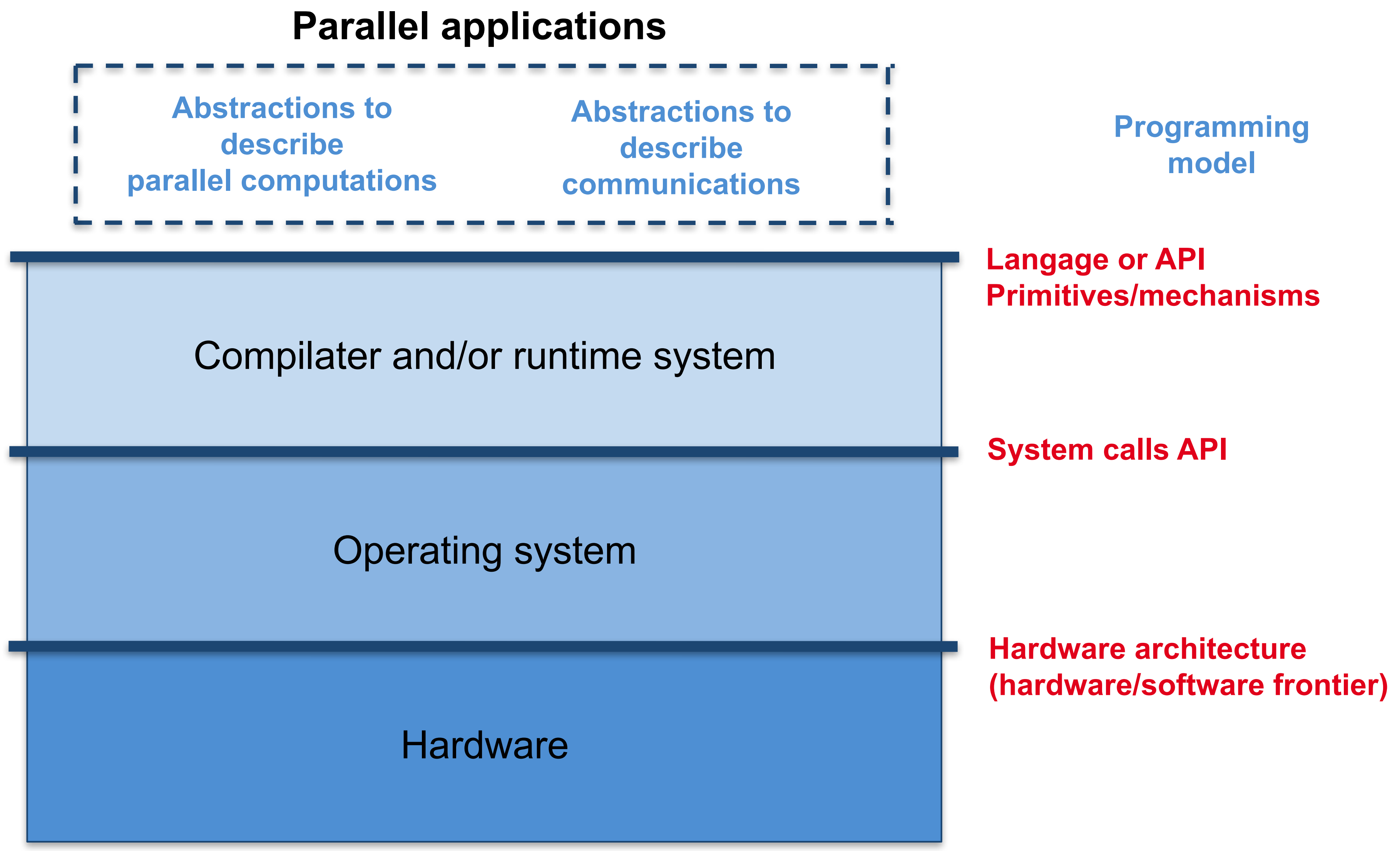
Frédéric Desprez

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


Parallel applications
Abstractions to describe parallel computations
Abstractions to describe communications
Programming model
Langage or API Primitives/mechanisms
Compilater and/or runtime system
System calls API
Operating system
Hardware architecture (hardware/software frontier)
Hardware

Concepts System abstraction System implementation

Introduction

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

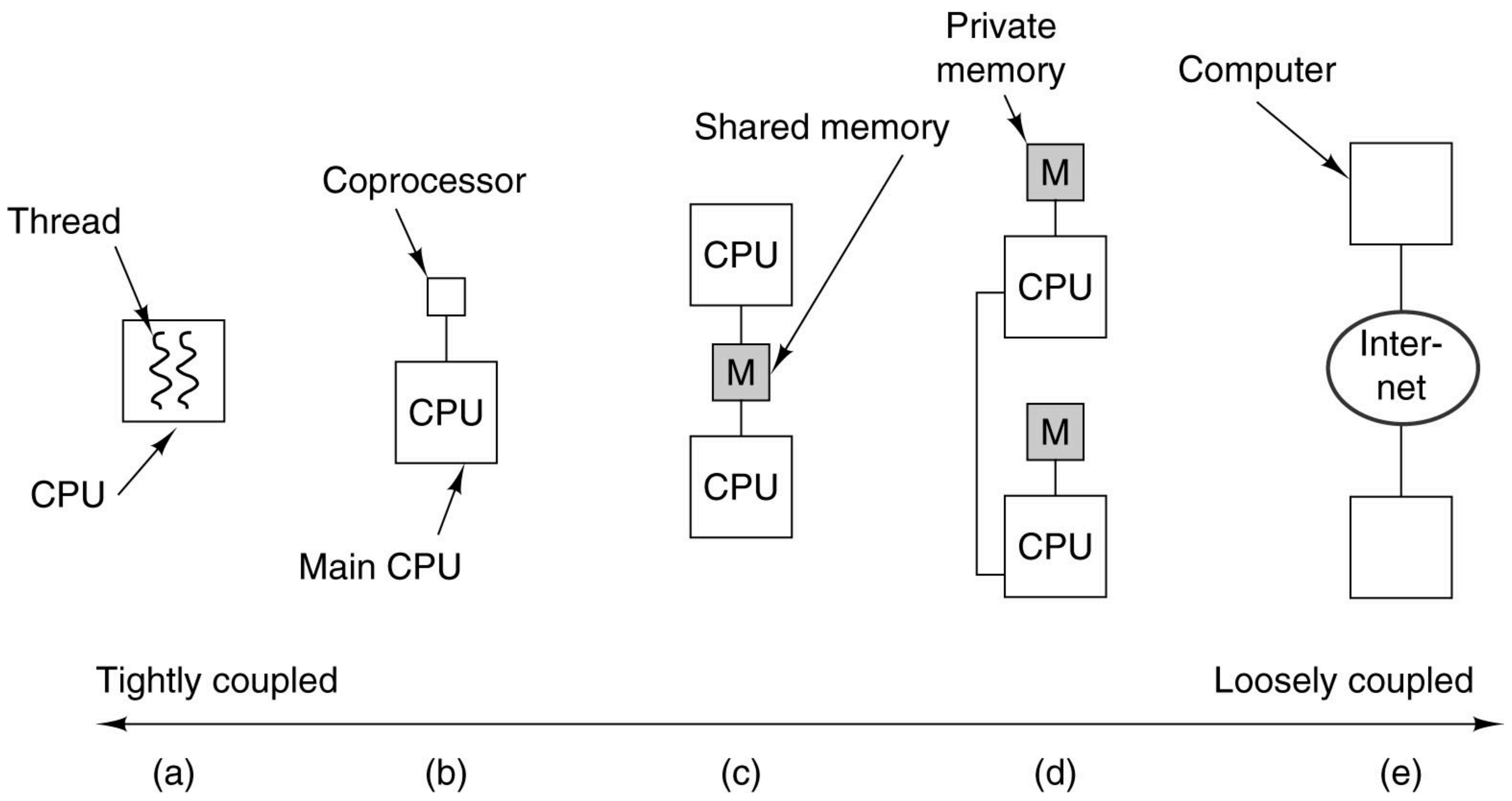
We will learn MPI (Message Passing Interface)

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

Other solution

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

Parallel architectures



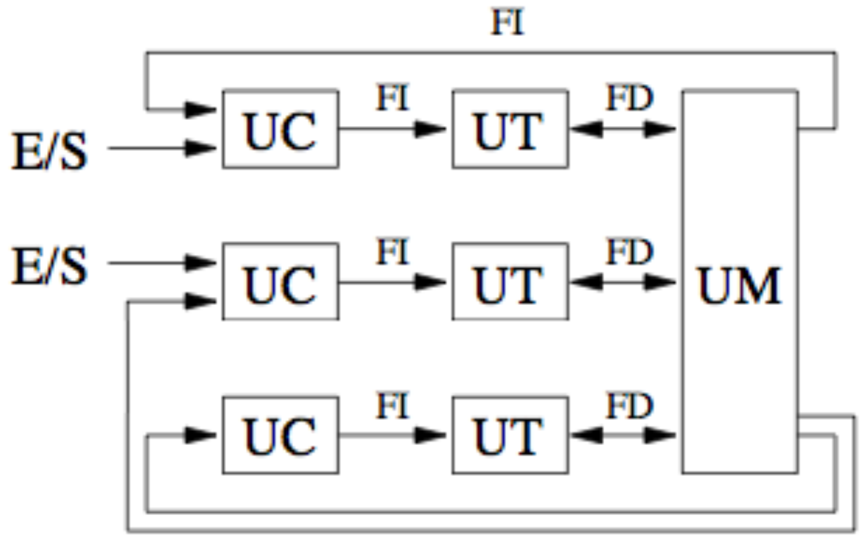
MIMD: Multiple Instructions stream, multiple data stream

Multi-Processor Machines

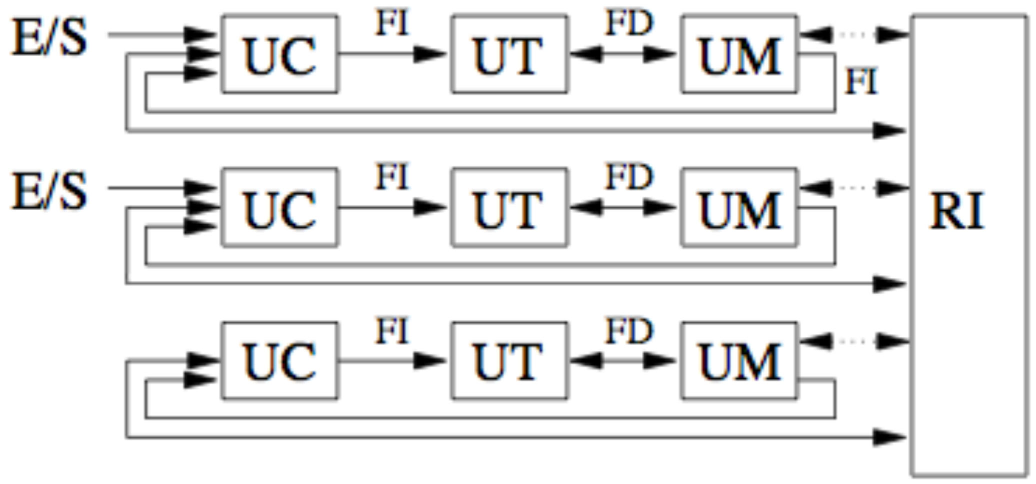
Each processor runs its own code asynchronously and independently

Two sub-classes

Shared memory



Distributed memory



A mix between SIMD and MIMD: SPMD (**Single Program, Multiple Data**)

Shared memory machine model

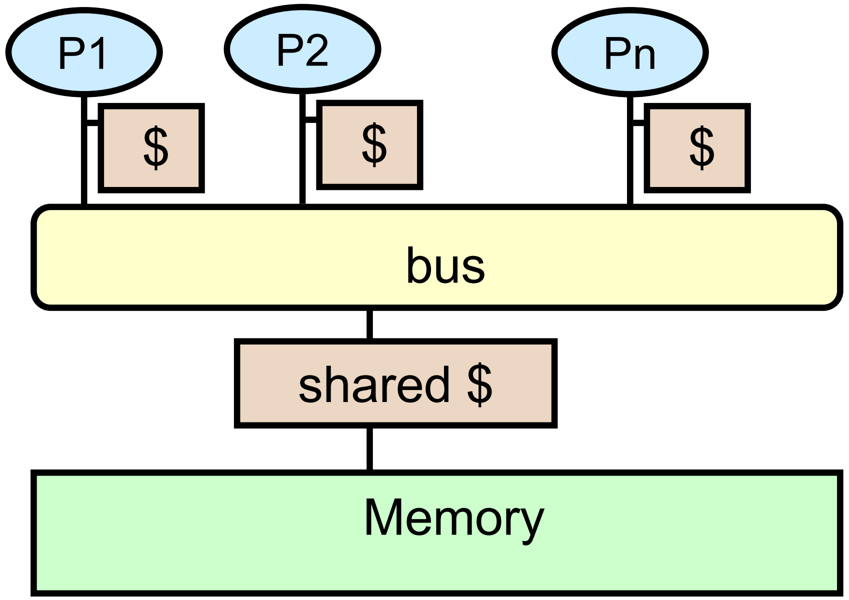
Processors are connected to a large shared memory

- 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

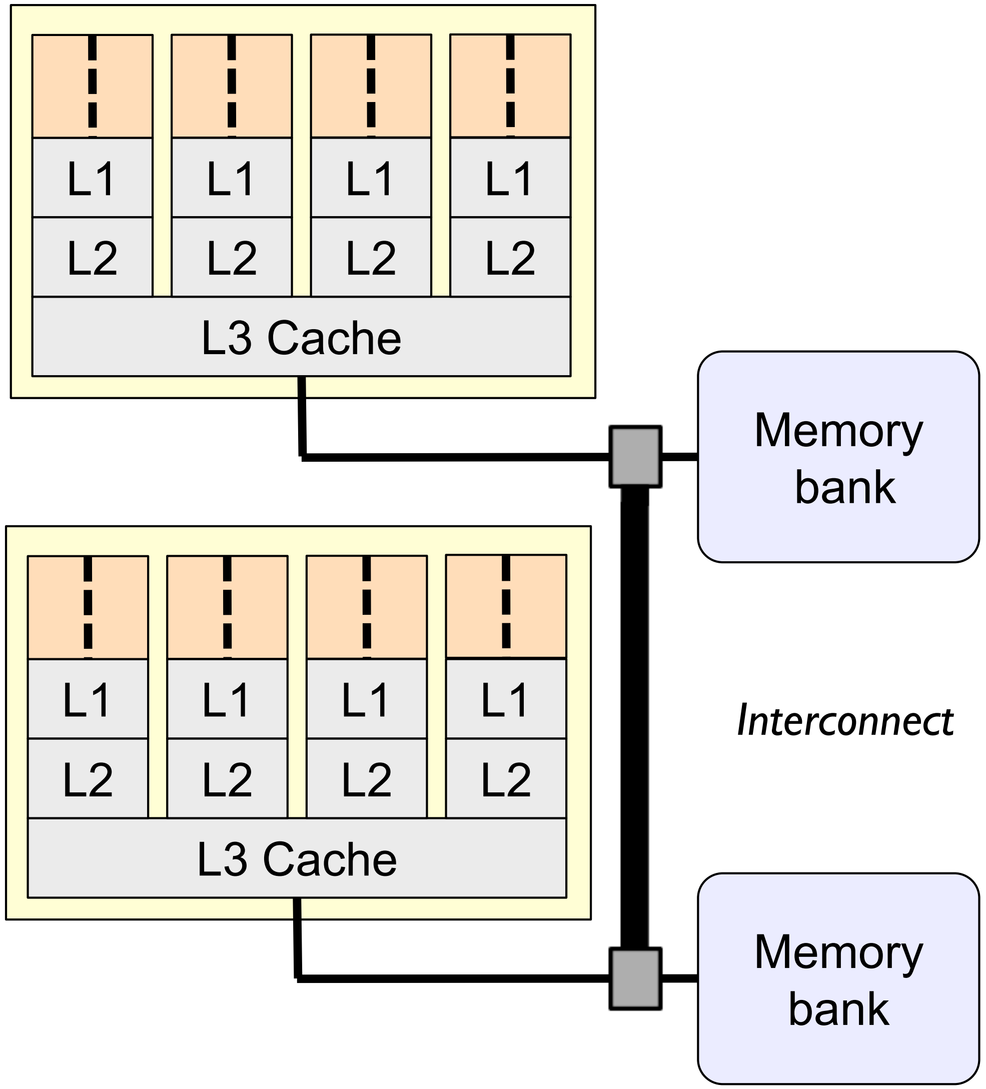


Note: $ = cache

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, CUDA, OpenACC, ...

OpenMP™

Multi-task programming model on shared memory architecture

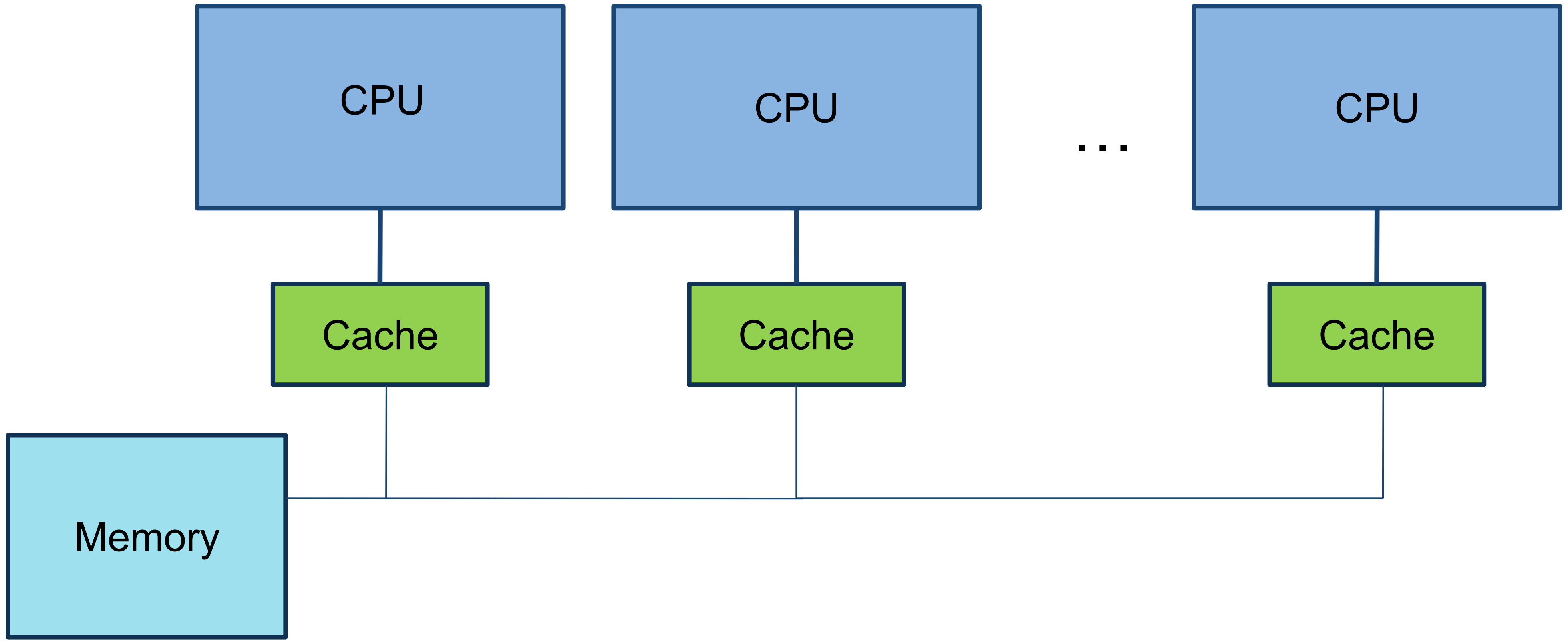
- Several tasks are executed in parallel
- Memory is shared (physically or virtually)
- Communication between tasks is done by reads and writes in the shared memory.
- Eg. The general-purpose multi-core processors share a common memory
  - Tasks can be assigned to distinct cores

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, ...

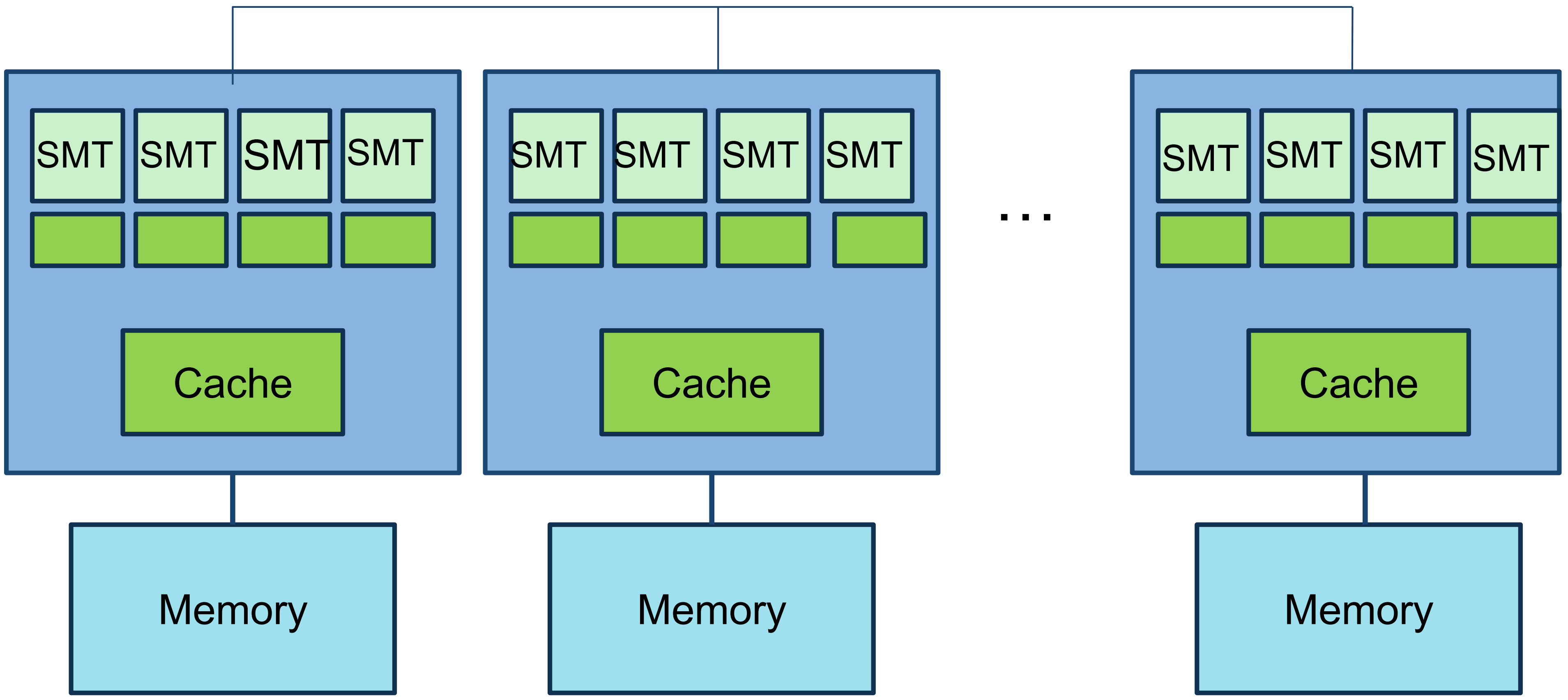
- OpenMP: a simpler alternative for the programmer

Multi-task programming on UMA architectures



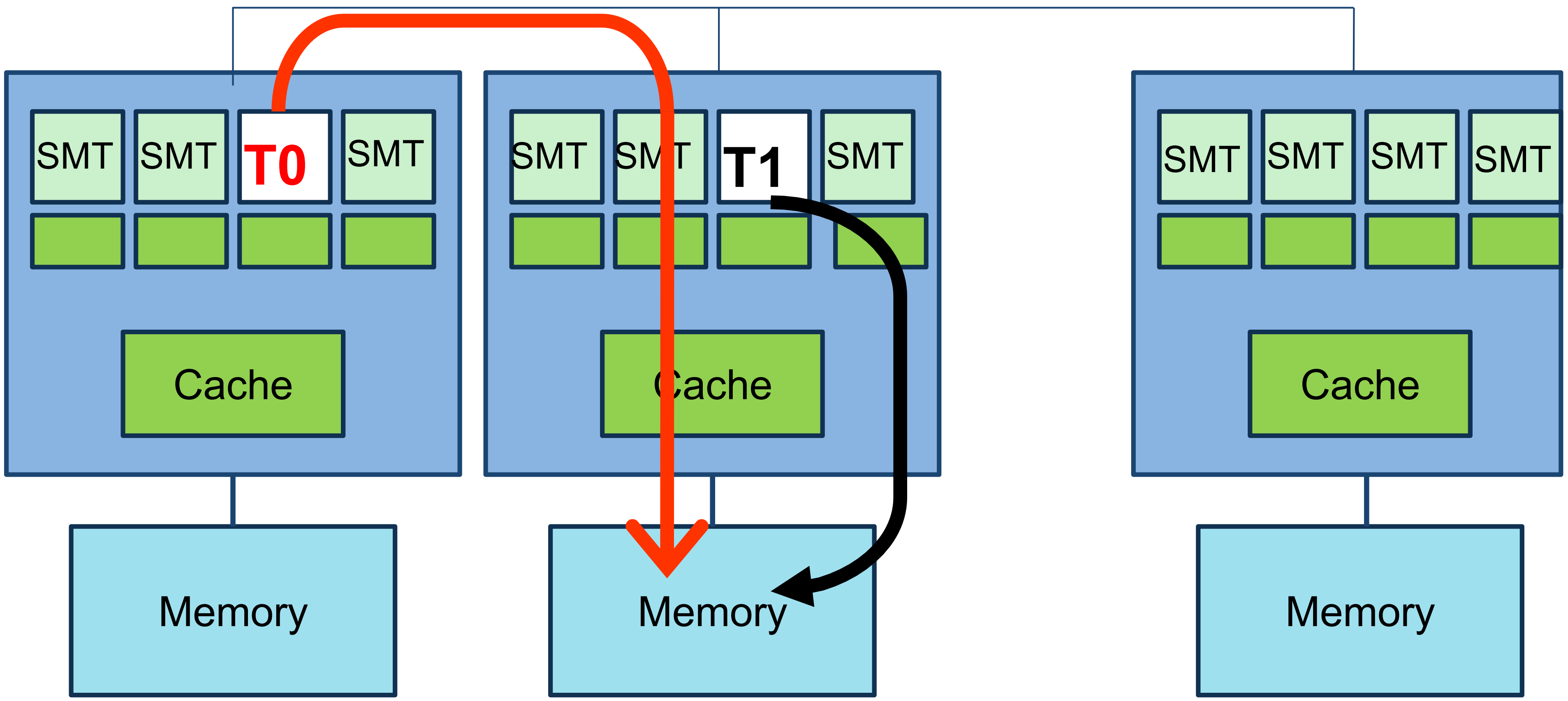
- Memory is shared
    - Uniform Memory Access Architectures (UMA)
    - An inherent problem: memory contentions

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)

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

Avantages

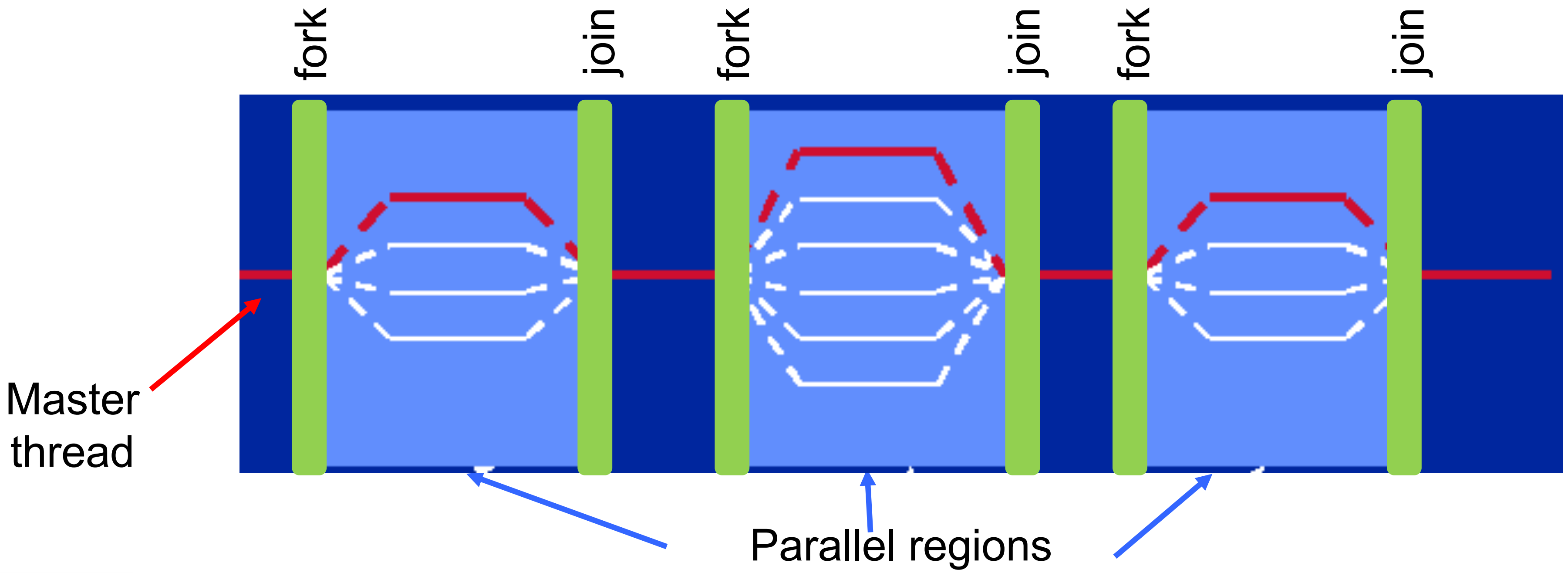
- Transparent and portable thread management
- Easy programming

Drawbacks

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

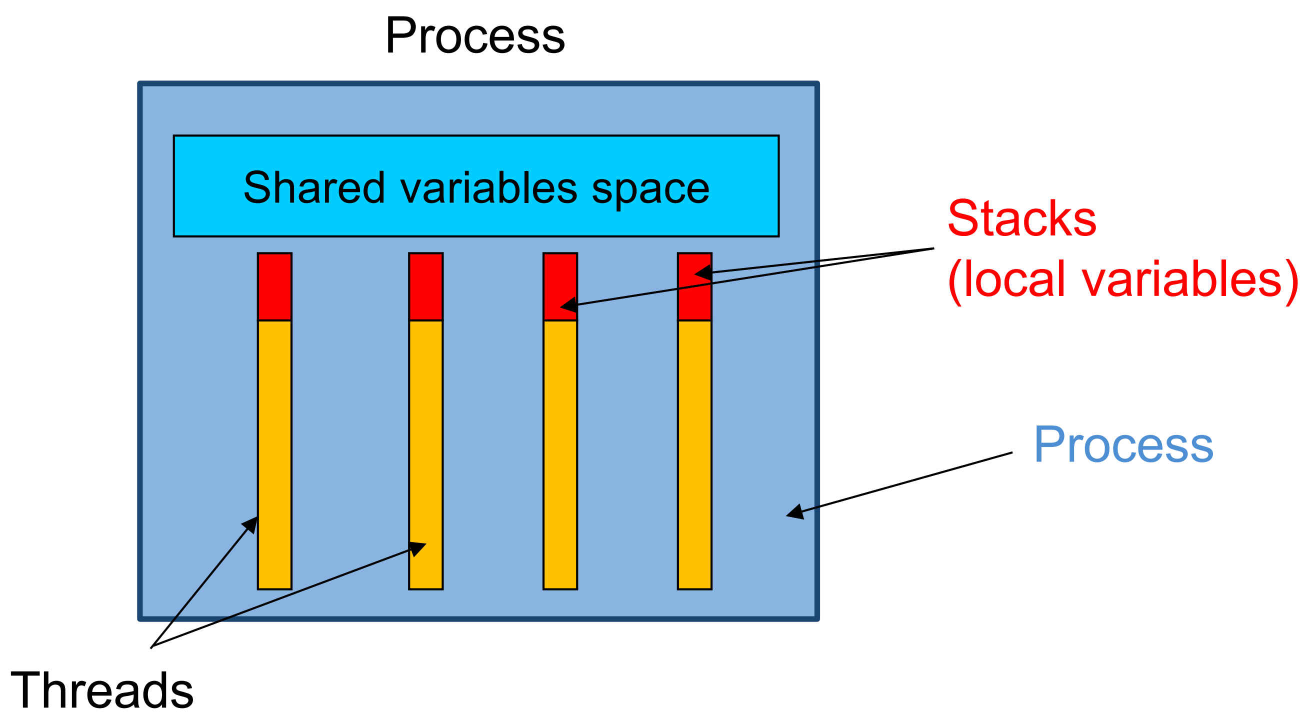
Introduction: execution model

- 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**



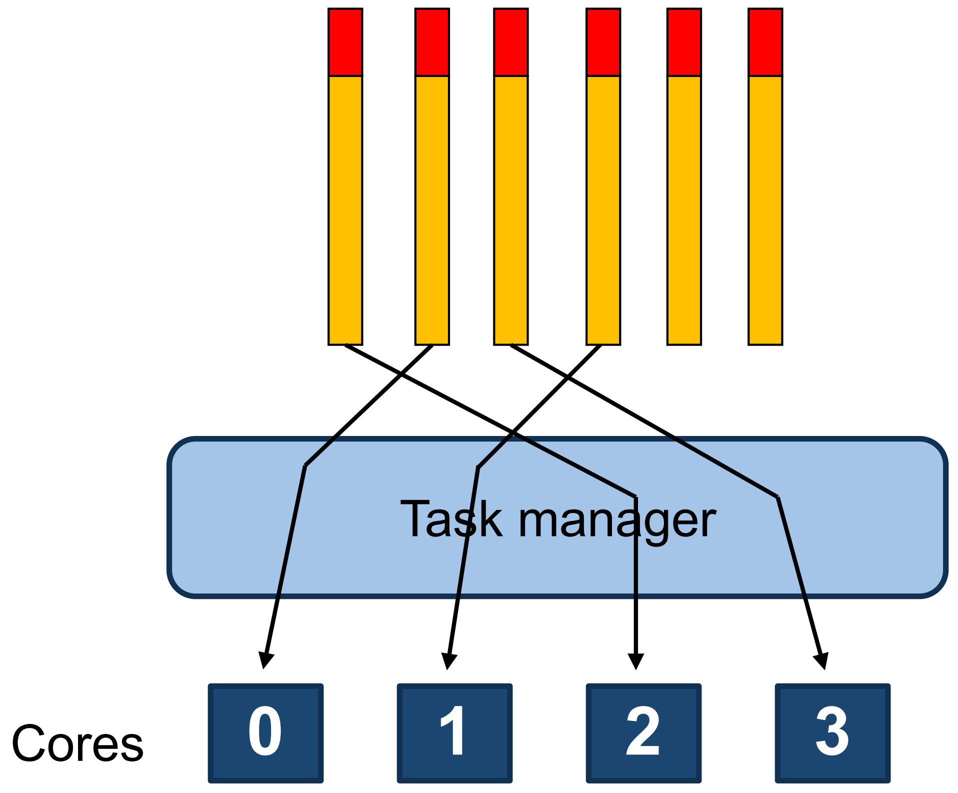
Introduction: threads

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

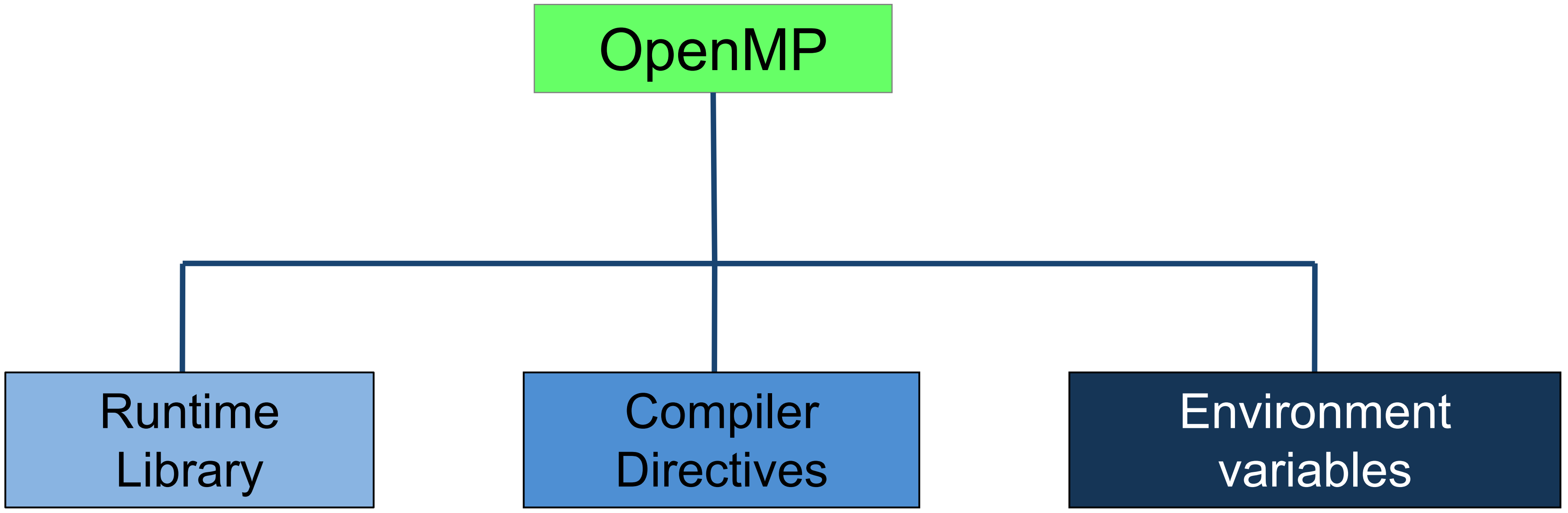


Introduction: execution of an OpenMP program on a multicore

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

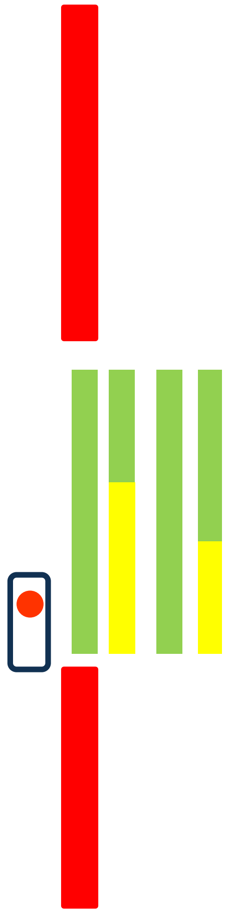
Fortran

```
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 sequentiel  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 sequentiel  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 !

OpenMP include file

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

Parallel region with default number of threads

Runtime library function to return a thread ID.

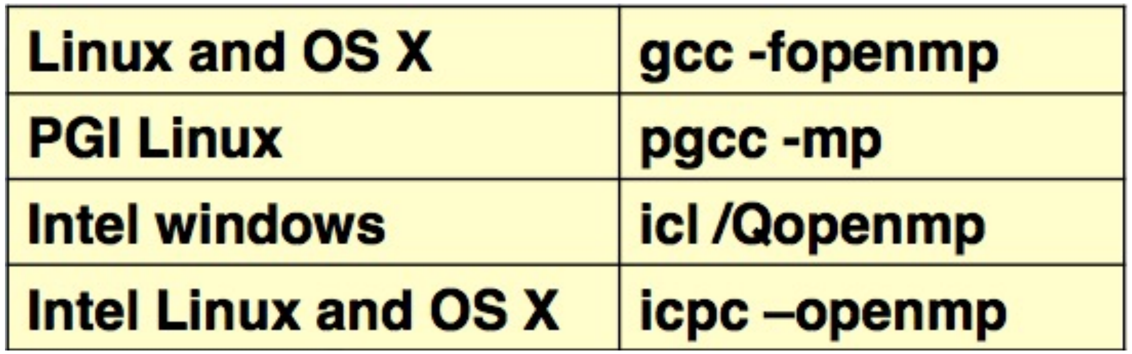
End of the Parallel region

Sample Output

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

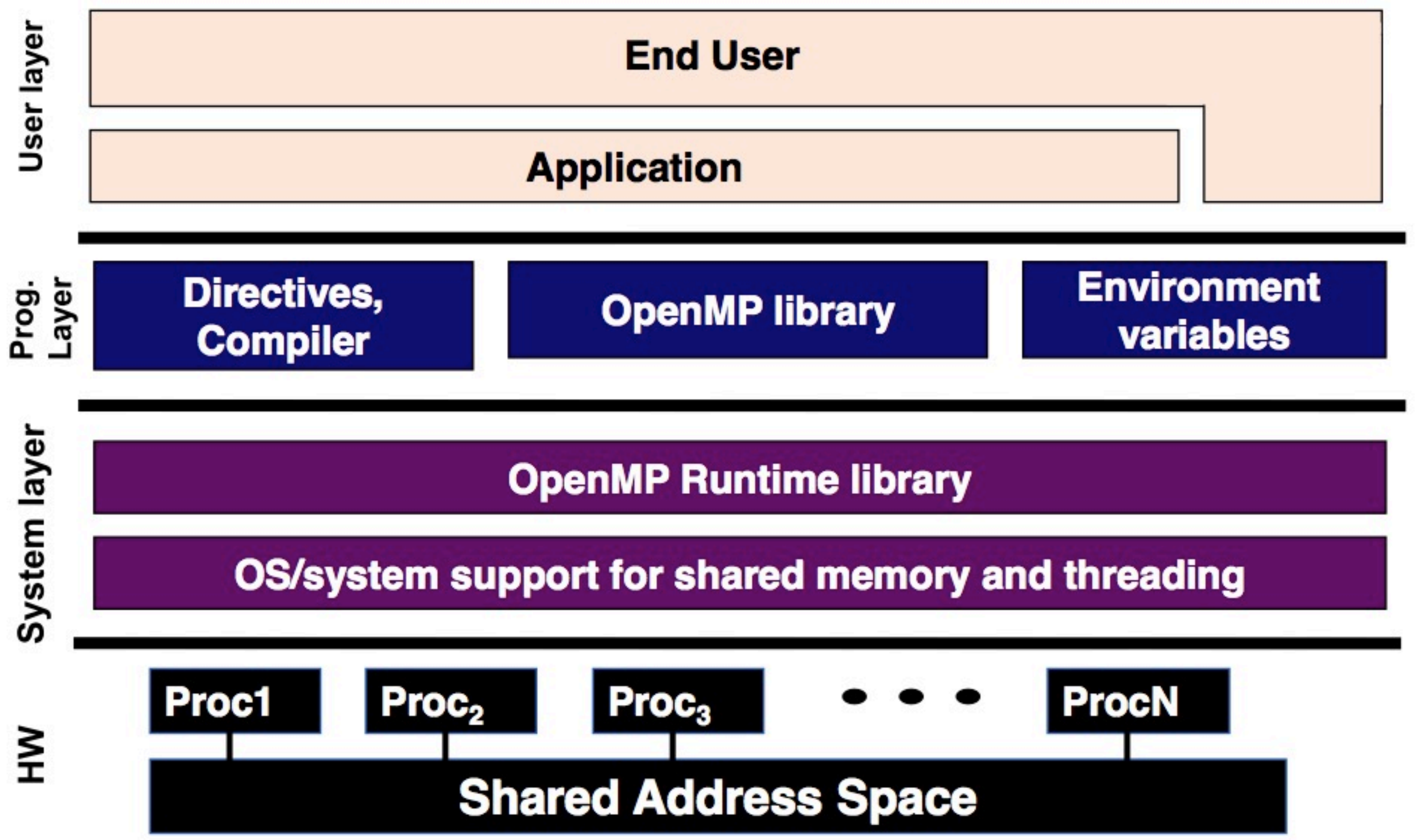
Example: Hello world Solution Calling the OpenMP compiler

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



Linux and OS X	gcc -fopenmp
PGI Linux	pgcc -mp
Intel windows	icl /Qopenmp
Intel Linux and OS X	icpc –openmp

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

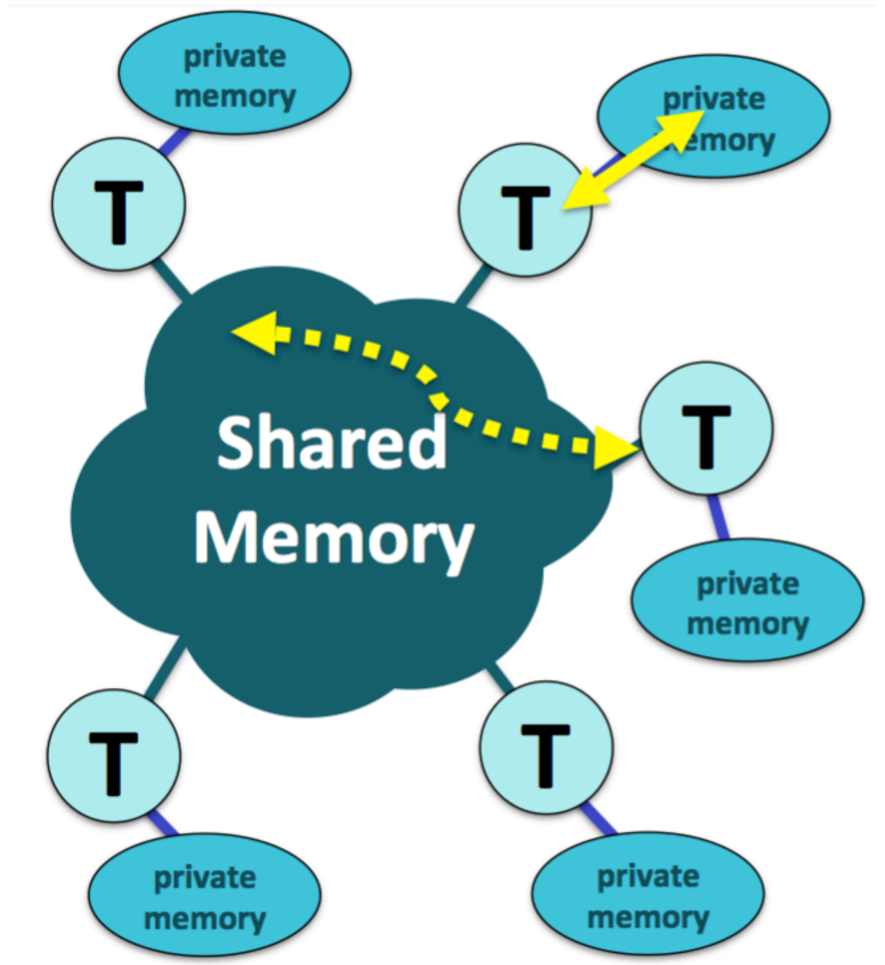
Threads are numbered

- 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 thread of each thread. It’s value is undefined at the entrance of the zone

```
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
```

- 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

Putting Threads to Work: the Worksharing Constructs

```
void simple_loop(int N,
                 float *a,
                 float *b)
{
    int i;
    // i, N, a and b are shared by
     default
    #pragma omp parallel firstprivate(N)
    {
        // i is private by default
        #pragma omp for
        for (i = 1; i <= N; i++) {
            b[i] = (a[i] + a[i-1]) / 2.0;
        }
    }
}
```

- **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

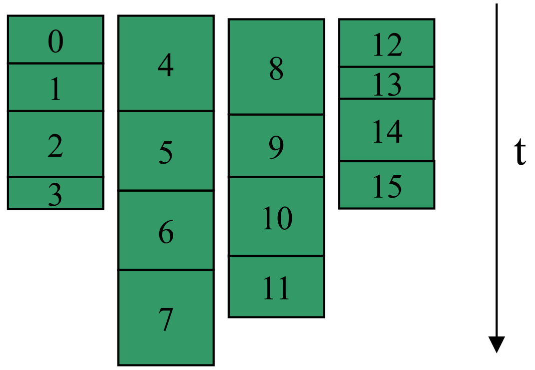
Beware: END PARALLEL DO includes a synchronization barrier!

Work sharing: SCHEDULE

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

By default packet-size = #_iterations/#_threads

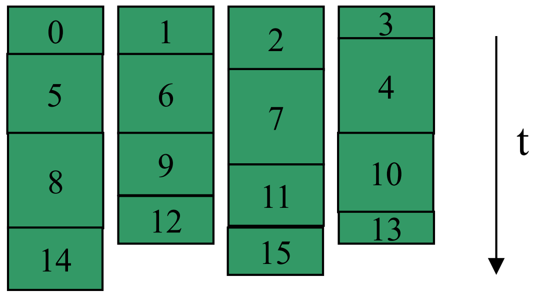
Ex: 16 iterations (0 to 15), 4 threads: packet size by default is 4



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

Packets are distributed to free threads in a dynamic way

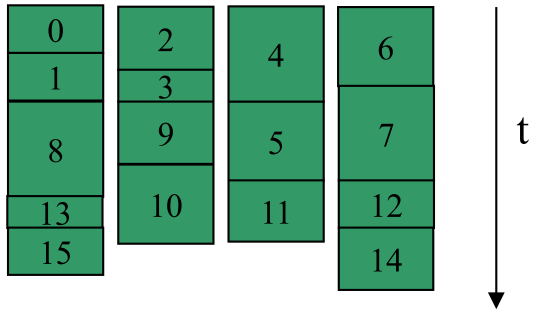
All the packets have the same size (except maybe the last one), by default the packet size is one



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

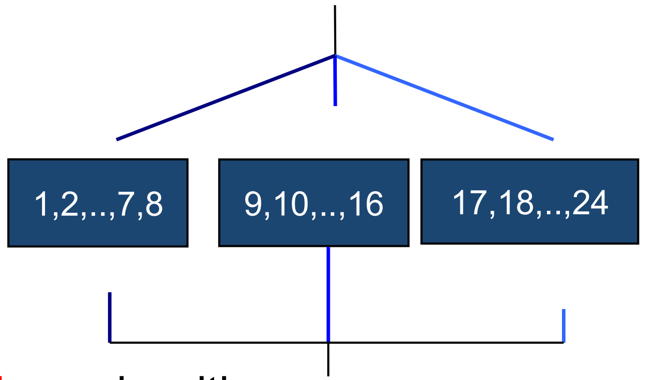
Packet-size: minimal packet size (1 by default) except the last one

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

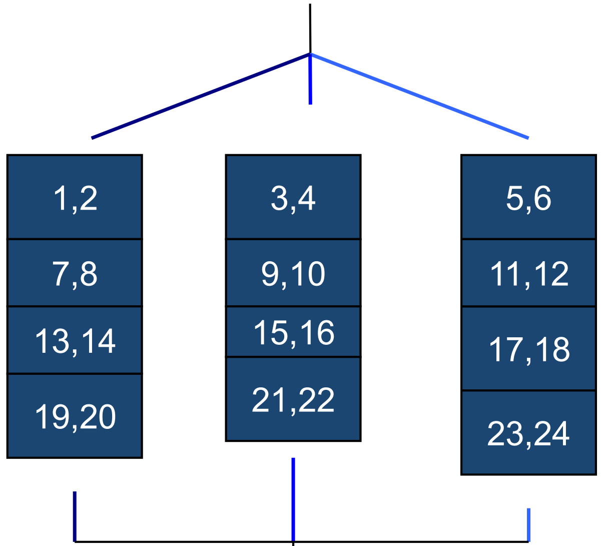


Work sharing: SCHEDULE

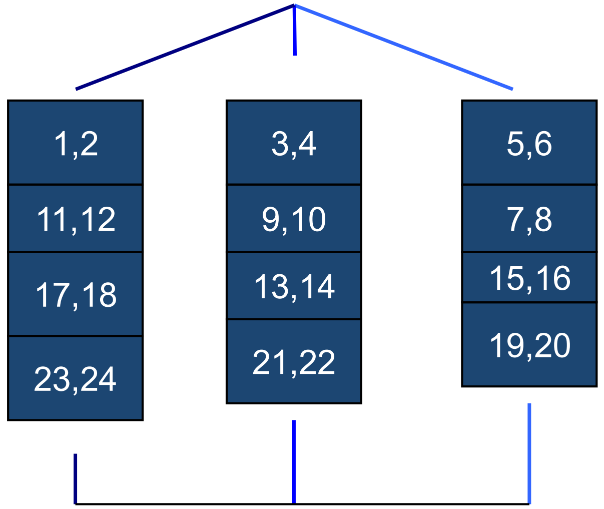
Ex: 24 iterations, 3 threads



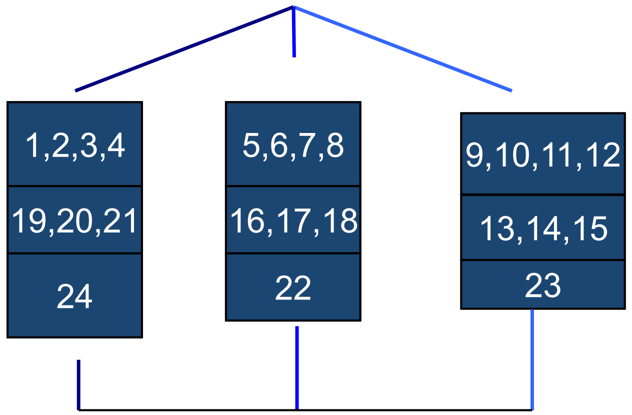
static mode with
Packet size = # iterations/# threads



Cyclic: static



Greedy: dynamic



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"
```

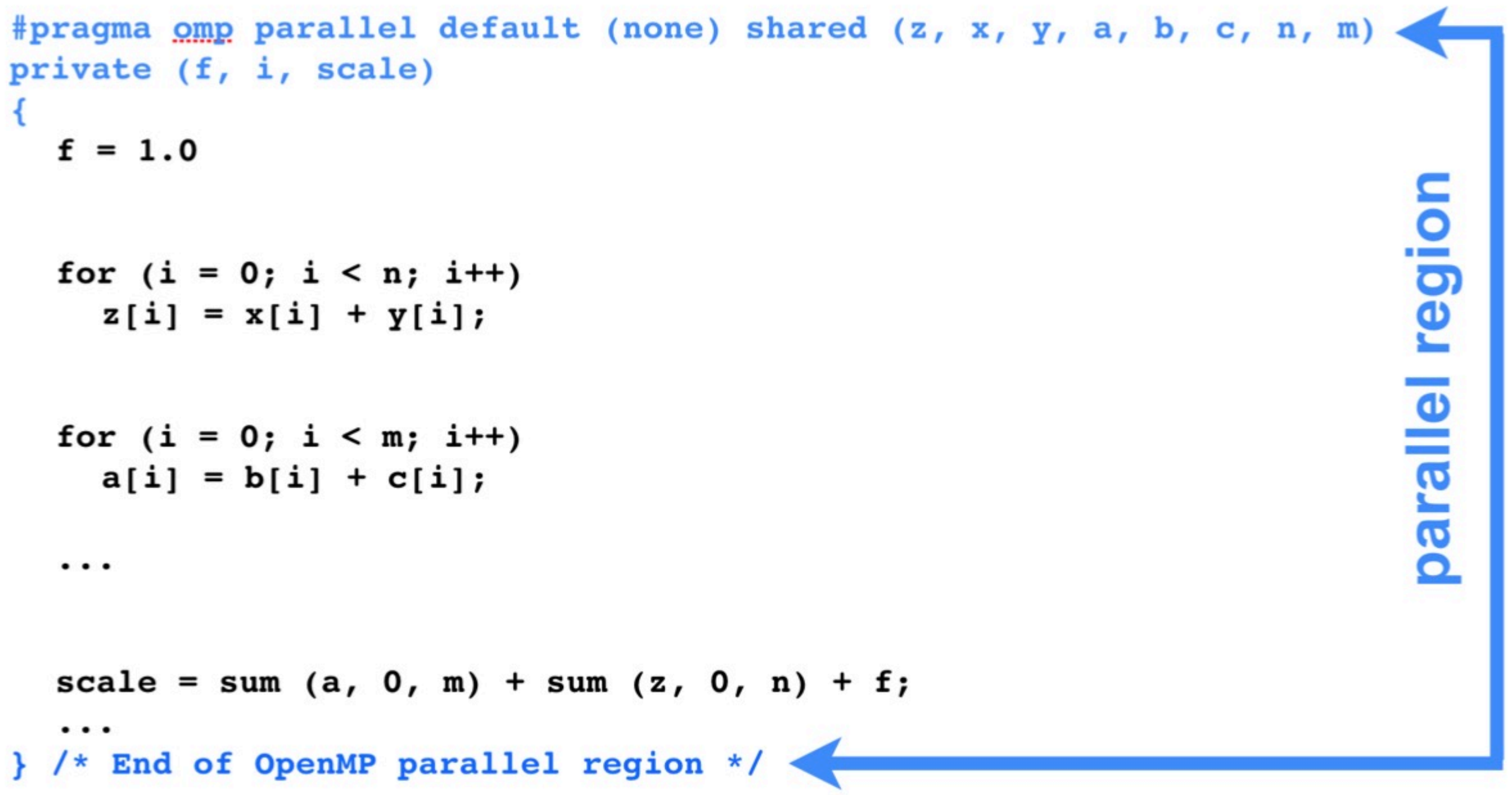
A first example to illustrate OpenMP capabilities

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

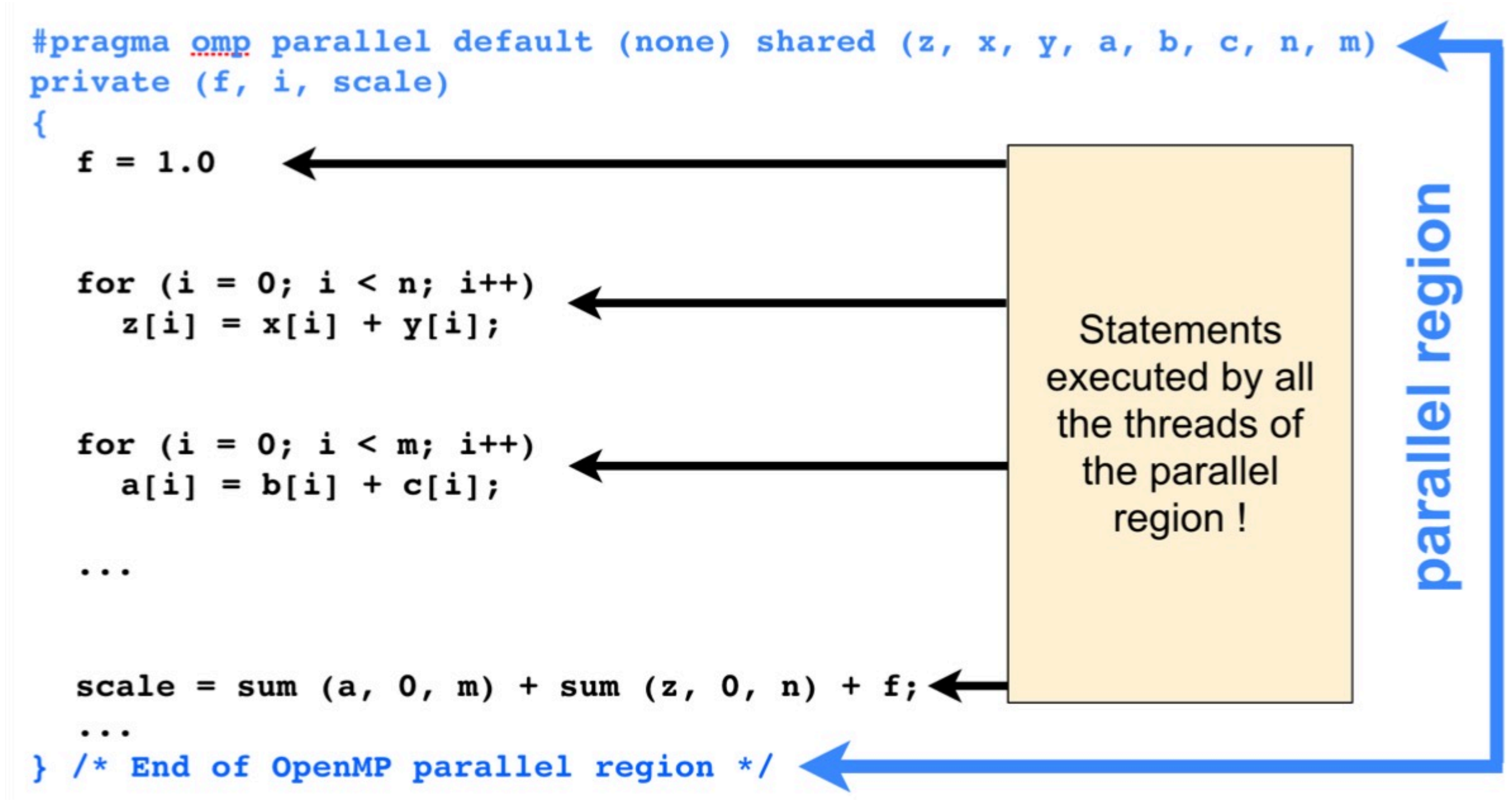
First create the parallel region and define the data-sharing attributes



```
#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 */
```

parallel region

A first example to illustrate OpenMP capabilities



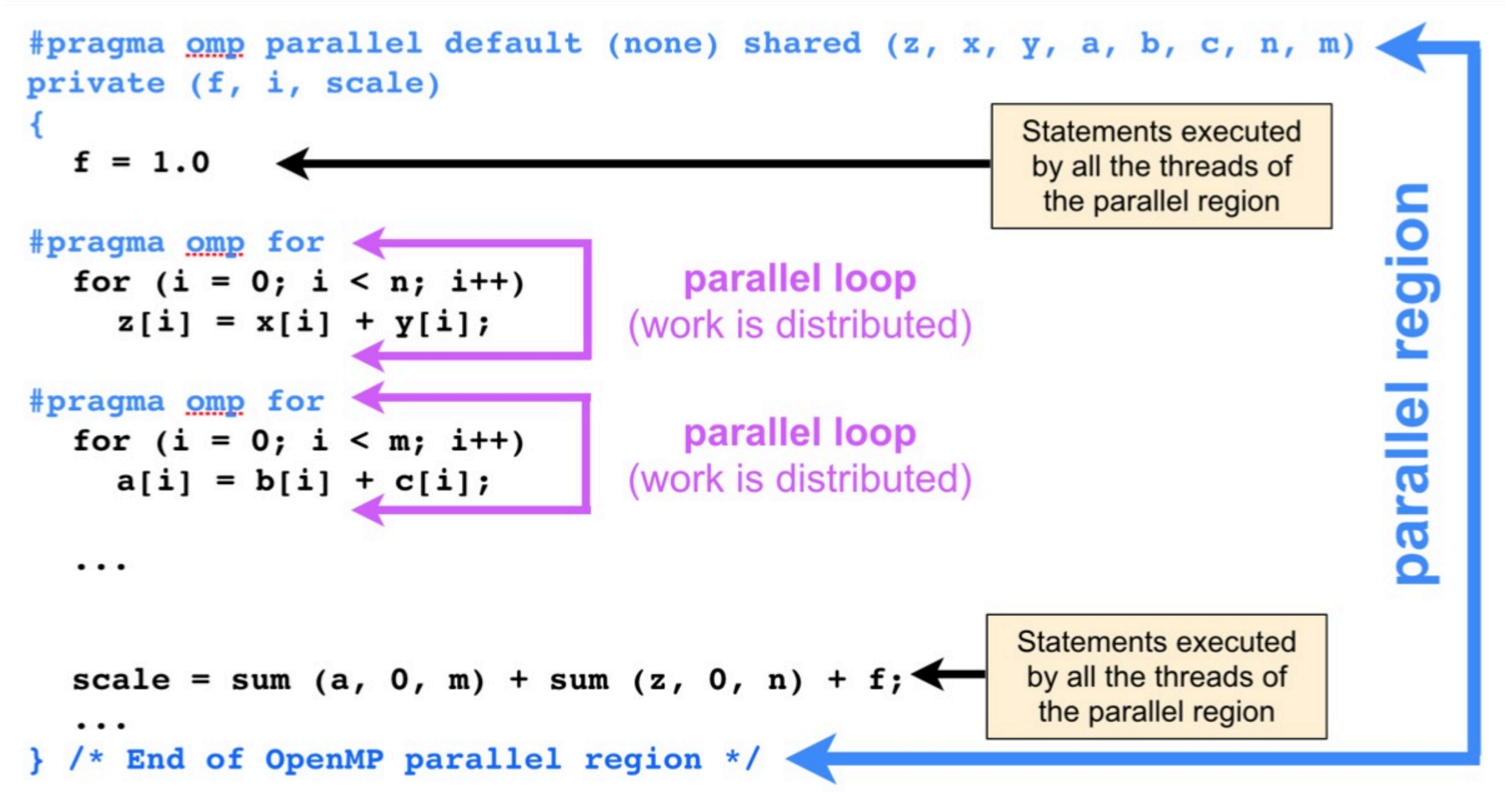
```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
   f = 1.0


   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 */
```

Statements executed by all the threads of the parallel region !
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

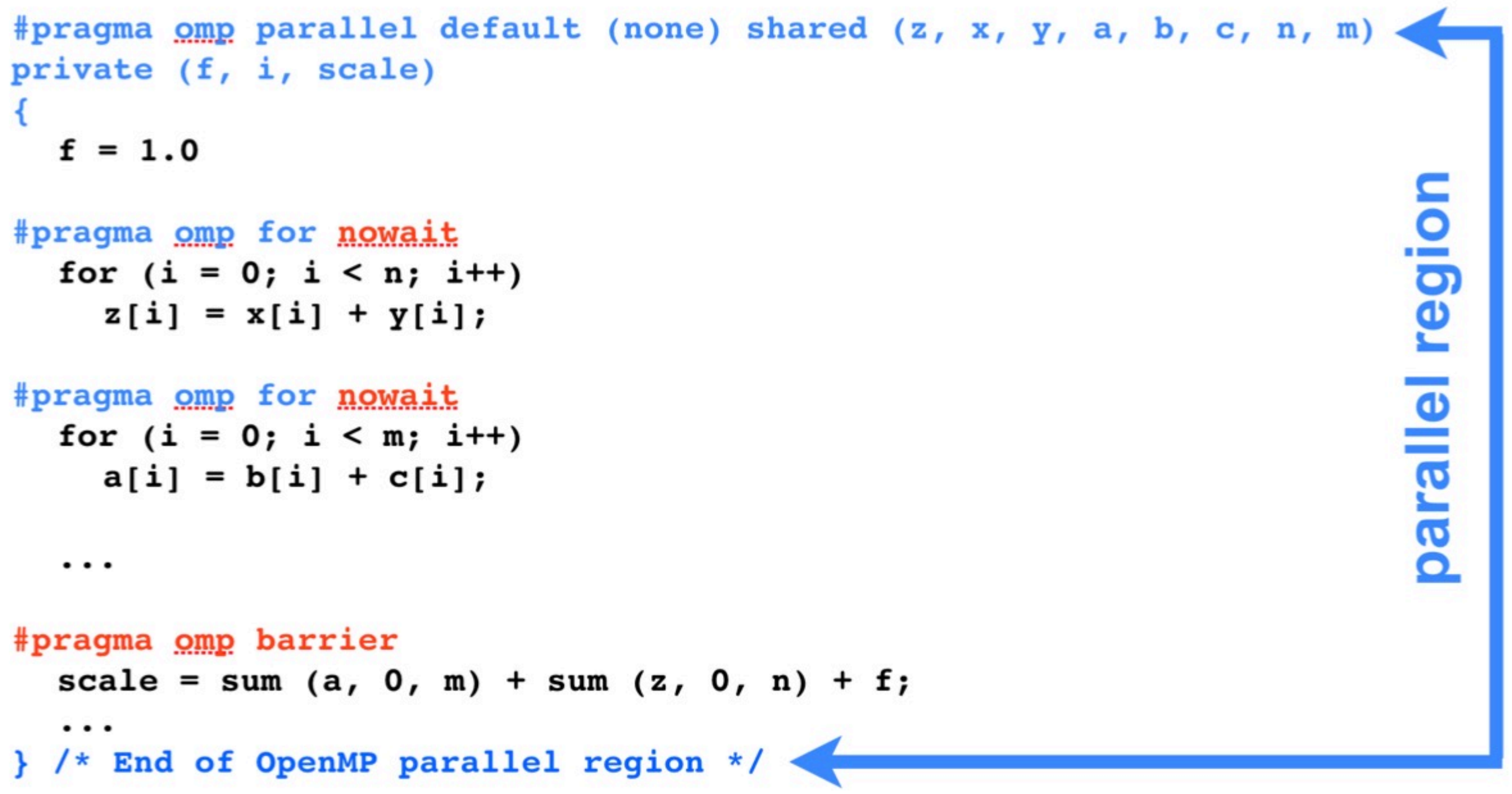


```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
  f = 1.0

#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)
Statements executed by all the threads of the parallel region
parallel region

Optimization #1: Remove Unnecessary Synchronizations

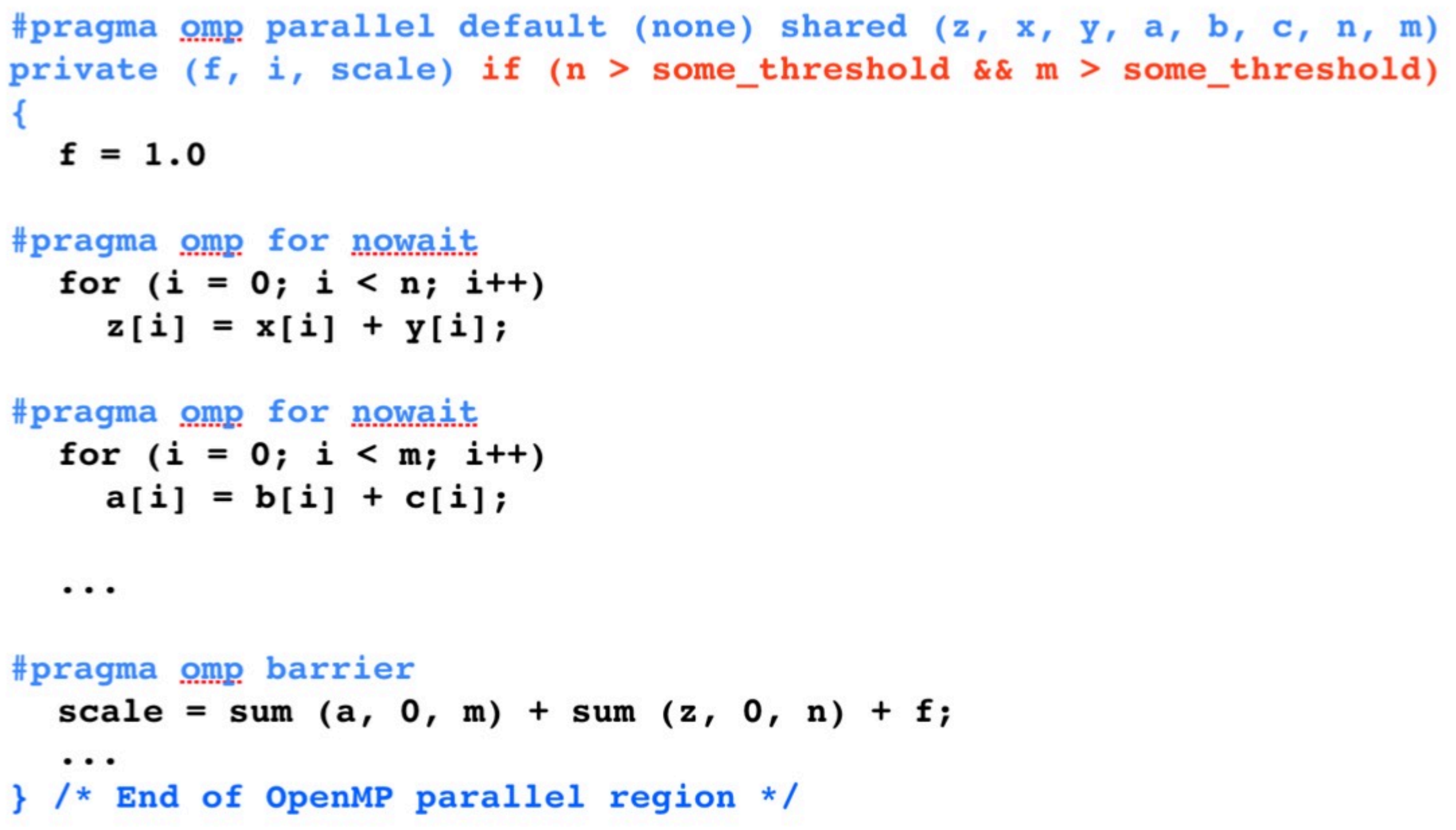
There are no dependencies between the two parallel loops, we remove the implicit barrier between the two



```
#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

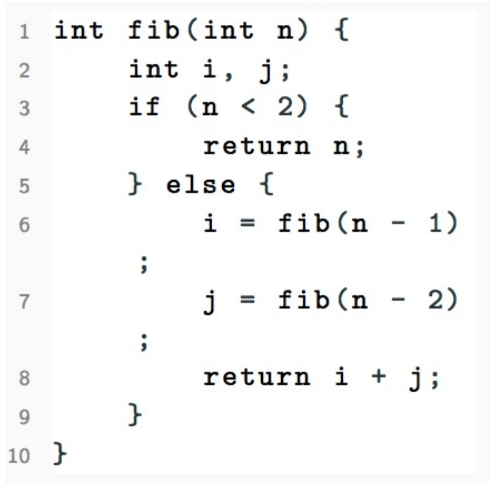
We don’t want to pay the price of thread management if the workload is too small to be computed in parallel



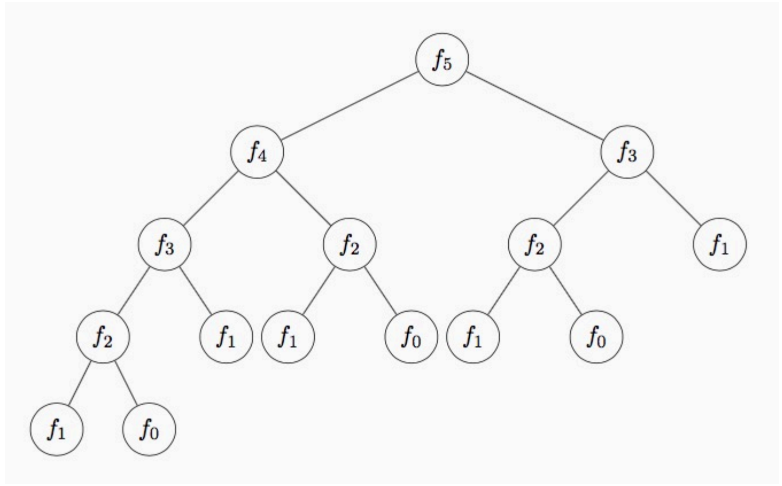
```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale) if (n > some_threshold && m > some_threshold)
{
  f = 1.0

#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 */
```

Extending the Scope of OpenMP with Task Parallelism

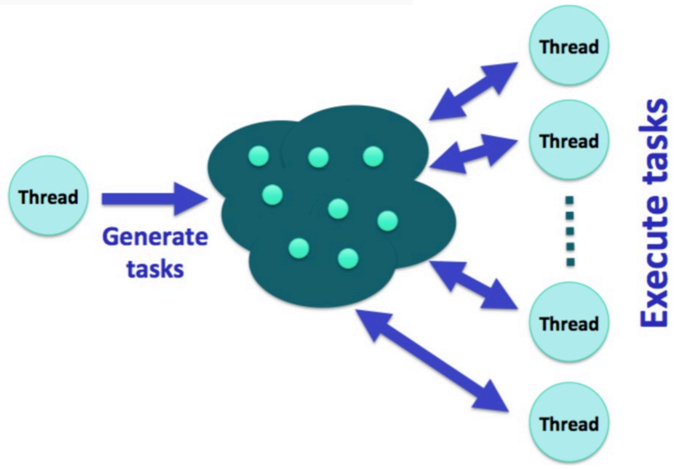
- **omp for** has made OpenMP popular and remains for most users its central feature. But what if my application was not written in a loop-based fashion?



```
int fib(int n) {
    int i, j;
    if (n < 2) {
        return n;
    } else {
        i = fib(n - 1)
    ;
        j = fib(n - 2)
    ;
        return i + j;
    }
}
```



- The OpenMP tasking concept : tasks generated by one OpenMP thread can be executed by any of the threads of the parallel region



Tasking in OpenMP: Basic Concept (cont'd)

- The application programmer specifies regions of code to be executed in a task with the **#pragma omp task** construct
- All tasks can be executed *independently*
- When any thread encounters a task construct, a task is generated
- Tasks are executed **asynchronously** by any thread of the parallel region
- Completion of the tasks can be guaranteed using the **taskwait** synchronization construct

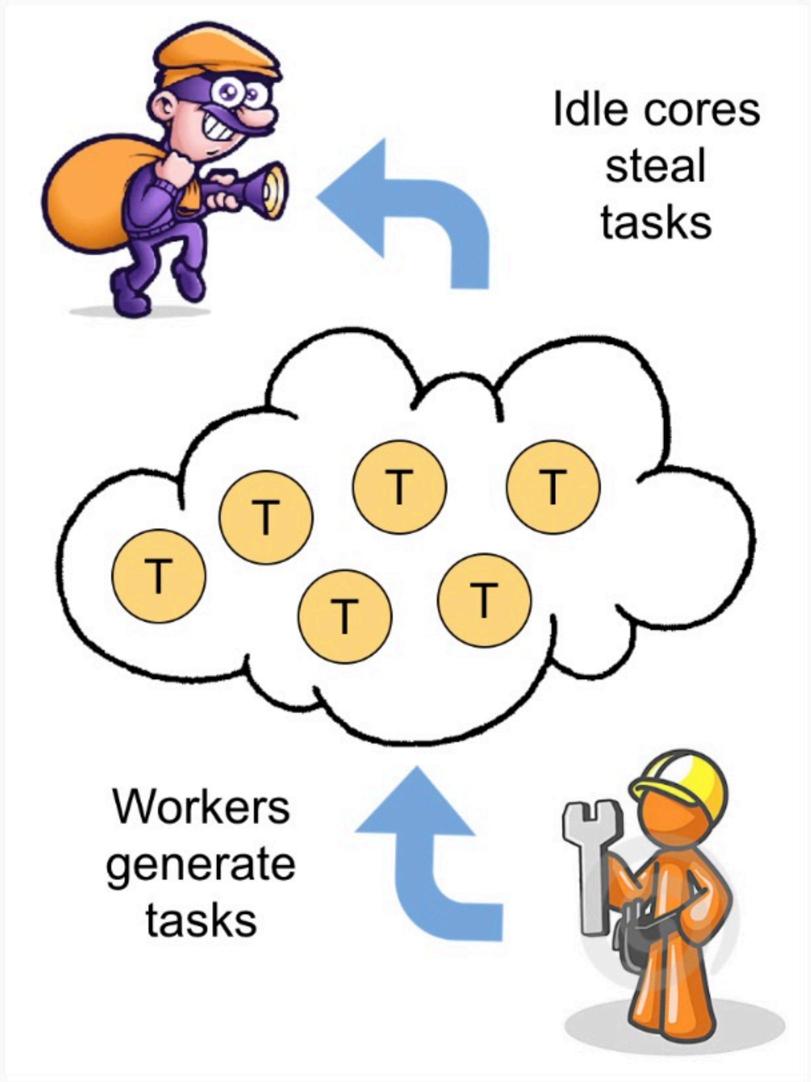
```
int main(void) {
    ...
    #pragma omp parallel
    {
        #pragma omp single
        res = fib(50);
    }
    ...
}

int fib(int n) {
    int i, j;
    if (n < 2) {
        return n;
    } else {
        #pragma omp task
        i = fib(n - 1);
        #pragma omp task
        j = fib(n - 2);
        #pragma omp task
     wait
        return i + j;
    }
}
```

Tasking in OpenMP: Execution Model

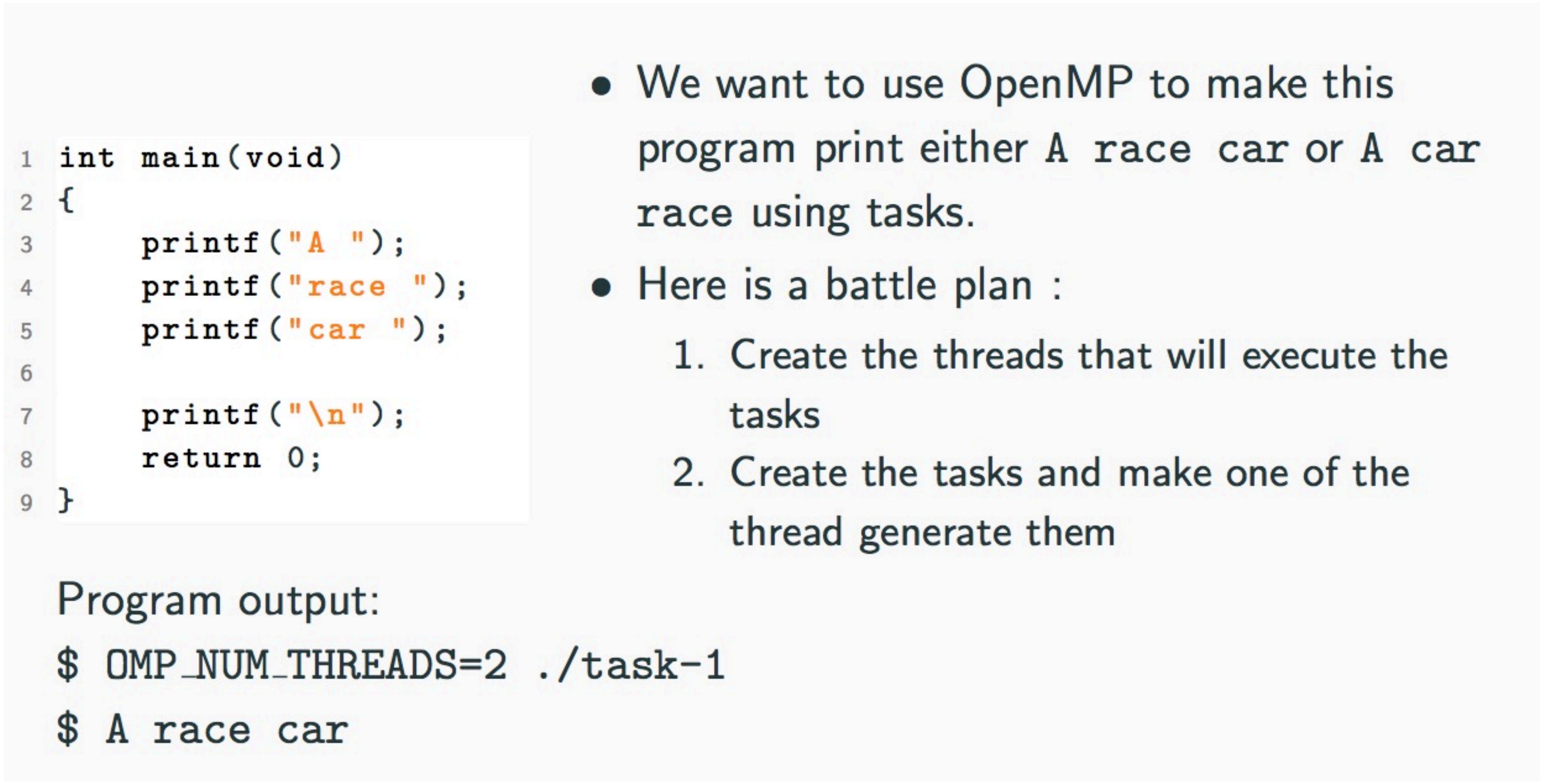
The Work-Stealing execution model

- Each thread has its own *task queue*
- Entering an `omp task` construct pushes a task to the thread's local queue
- When a thread's local queue is empty, it steals tasks from other queues

Tasks are well suited to applications with irregular workload.



First OpenMP Tasking Experience



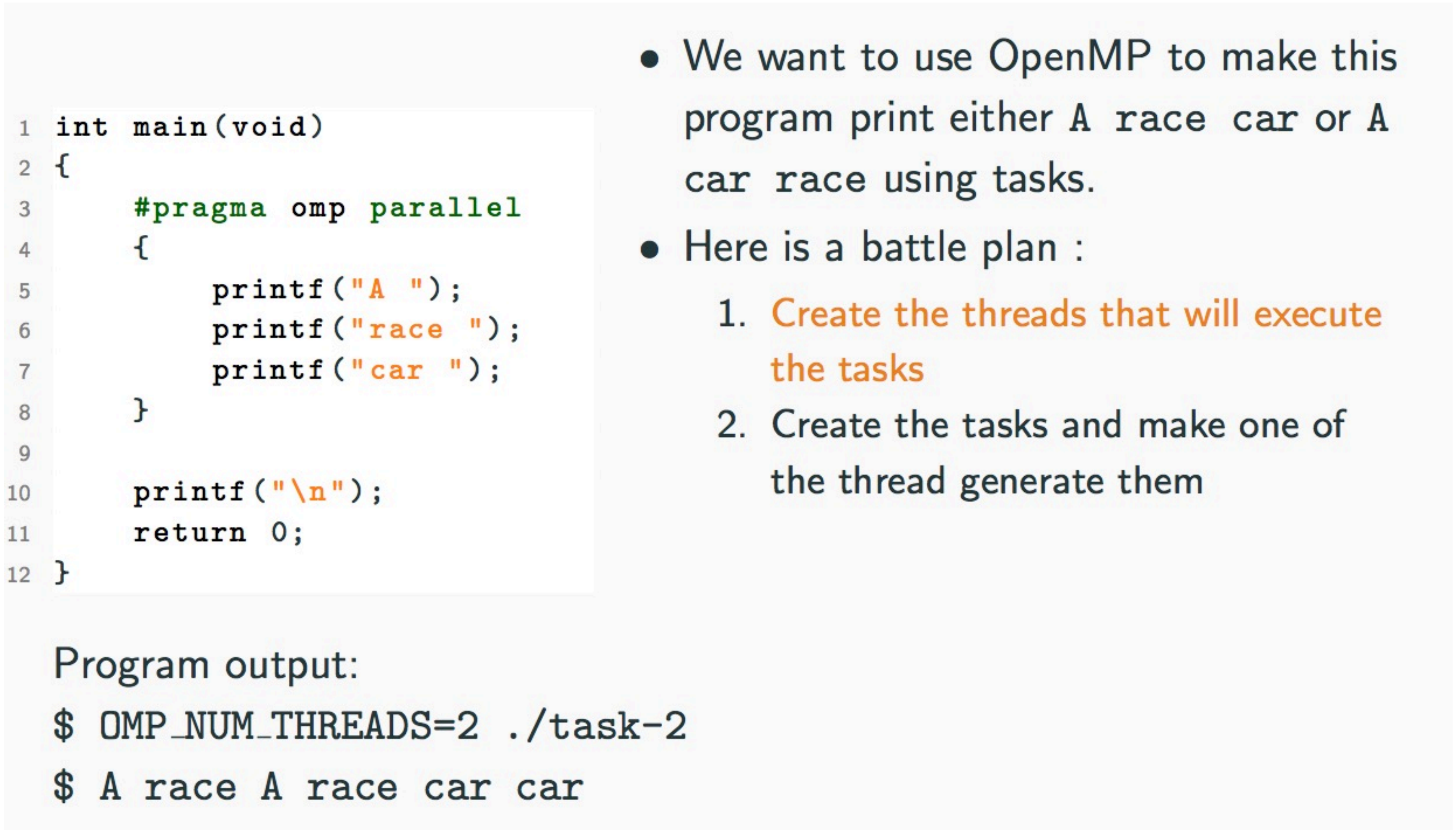
```c
int main(void)
{
    printf("A ");
    printf("race ");
    printf("car ");

    printf("\n");
    return 0;
}
```

- We want to use OpenMP to make this program print either `A race car` or `A car race` using tasks.
- Here is a battle plan :
  1. Create the threads that will execute the tasks
  2. Create the tasks and make one of the thread generate them

Program output:

```
$ OMP_NUM_THREADS=2 ./task-1
$ A race car
```

First OpenMP Tasking Experience, contd.



```c
int main(void)
{
    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");
    }

    printf("\n");
    return 0;
}
```

- We want to use OpenMP to make this program print either `A race car` or `A car race` using tasks.
- Here is a battle plan :
  1. Create the threads that will execute the tasks
  2. Create the tasks and make one of the thread generate them

Program output:

```
$ OMP_NUM_THREADS=2 ./task-2
$ A race A race car car
```

First OpenMP Tasking Experience, contd.

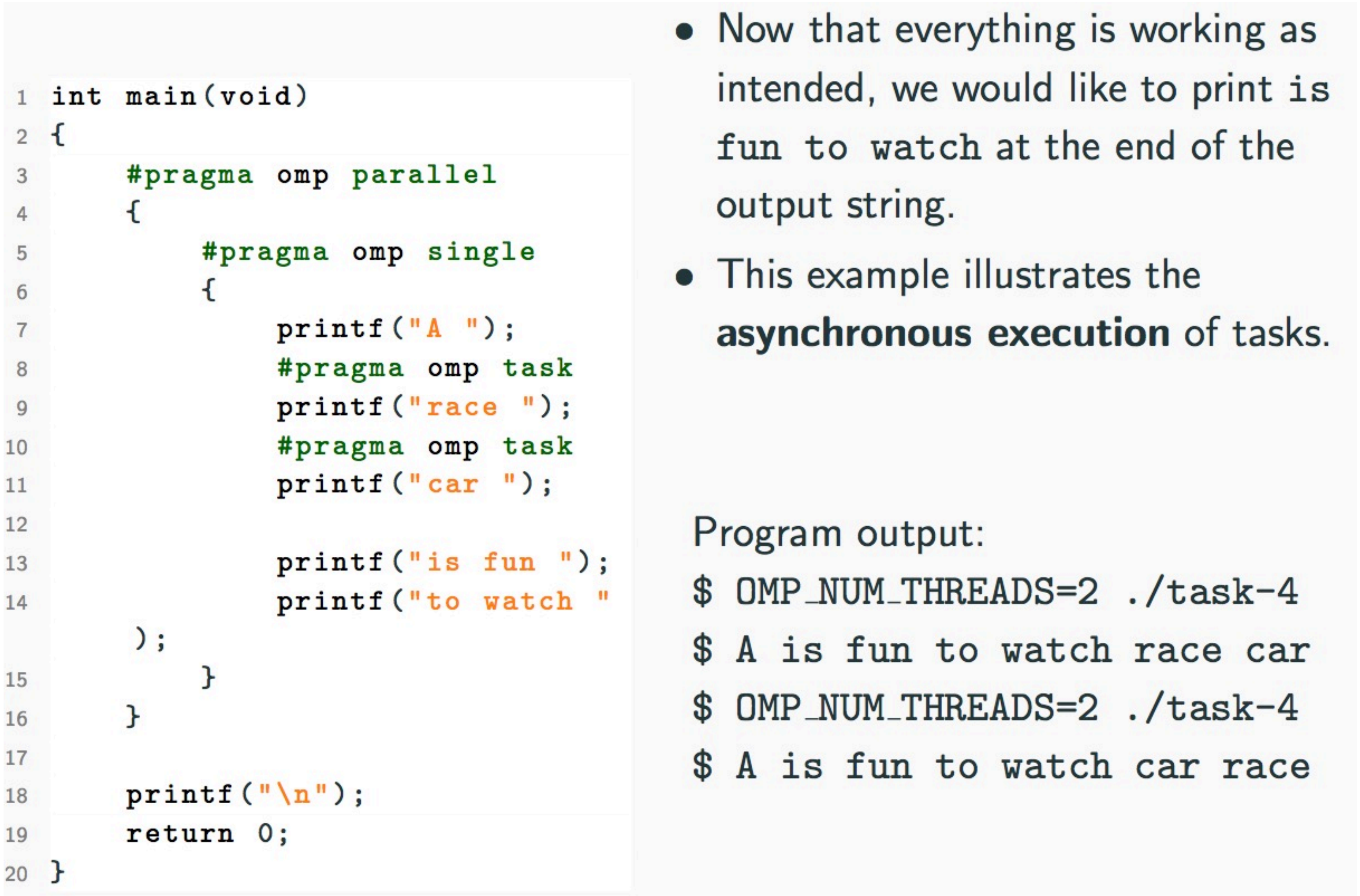
```
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp
     task
            printf("race ")
    ;
            #pragma omp
     task
            printf("car ");
        }
    }

    printf("\n");
    return 0;
}
```

- We want to use OpenMP to make this program print either `A race car` or `A car race` using tasks.
- Here is a battle plan :
  1. Create the threads that will execute the tasks
  2. Create the tasks and make one of the thread generate them

Program output:

```
$ OMP_NUM_THREADS=2 ./task-3
$ A race car
$ OMP_NUM_THREADS=2 ./task-3
$ A car race
```

First OpenMP Tasking Experience, contd.



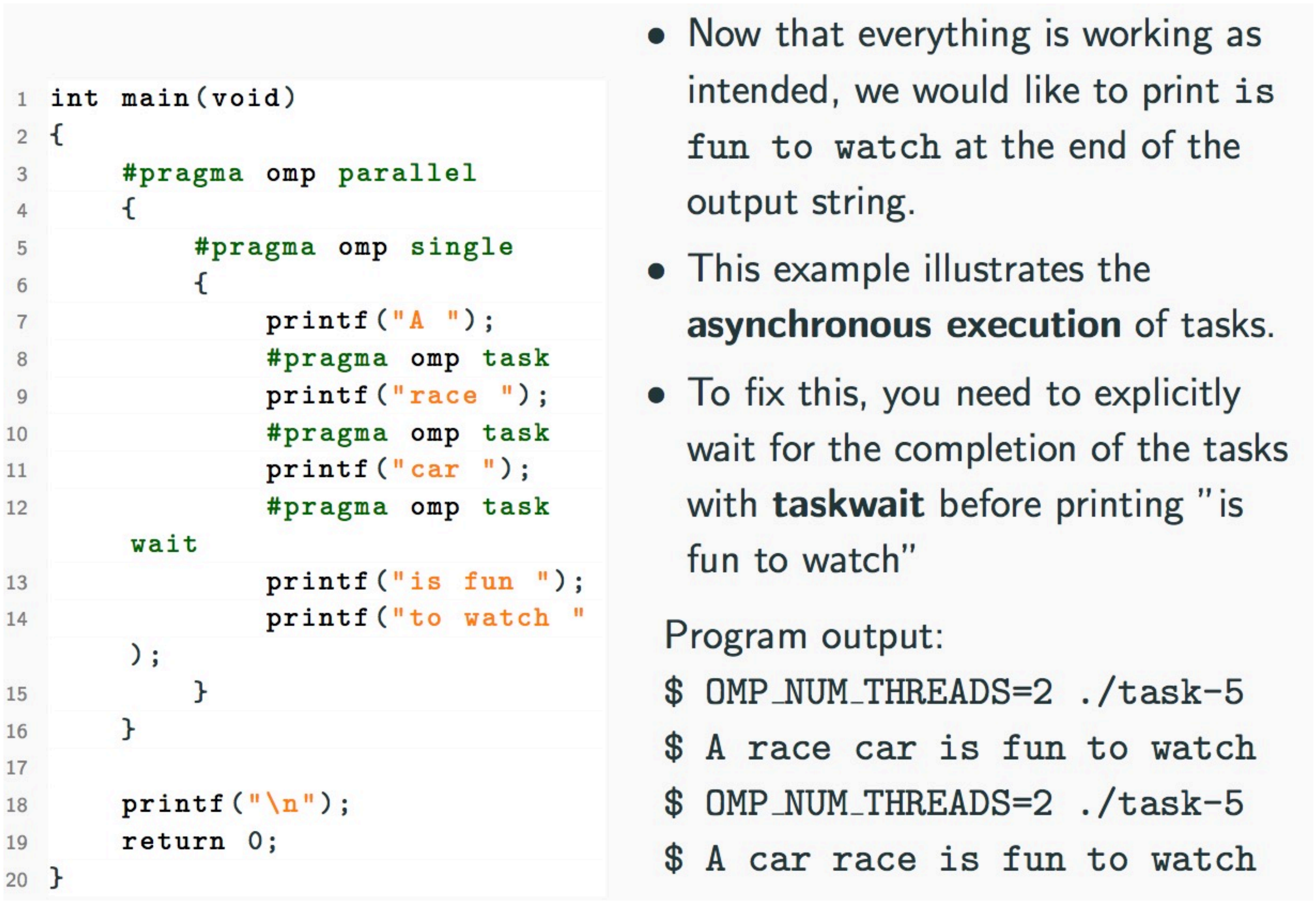
```c
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            printf("race ");
            #pragma omp task
            printf("car ");

            printf("is fun ");
            printf("to watch ");
        }
    }

    printf("\n");
    return 0;
}
```

- Now that everything is working as intended, we would like to print is `fun to watch` at the end of the output string.
- This example illustrates the **asynchronous execution** of tasks.

Program output:

```
$ OMP_NUM_THREADS=2 ./task-4
$ A is fun to watch race car
$ OMP_NUM_THREADS=2 ./task-4
$ A is fun to watch car race
```

First OpenMP Tasking Experience, contd.



```
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            printf("race ");
            #pragma omp task
            printf("car ");
            #pragma omp task
     wait
            printf("is fun ");
            printf("to watch "
    );
        }
    }

    printf("\n");
    return 0;
}
```

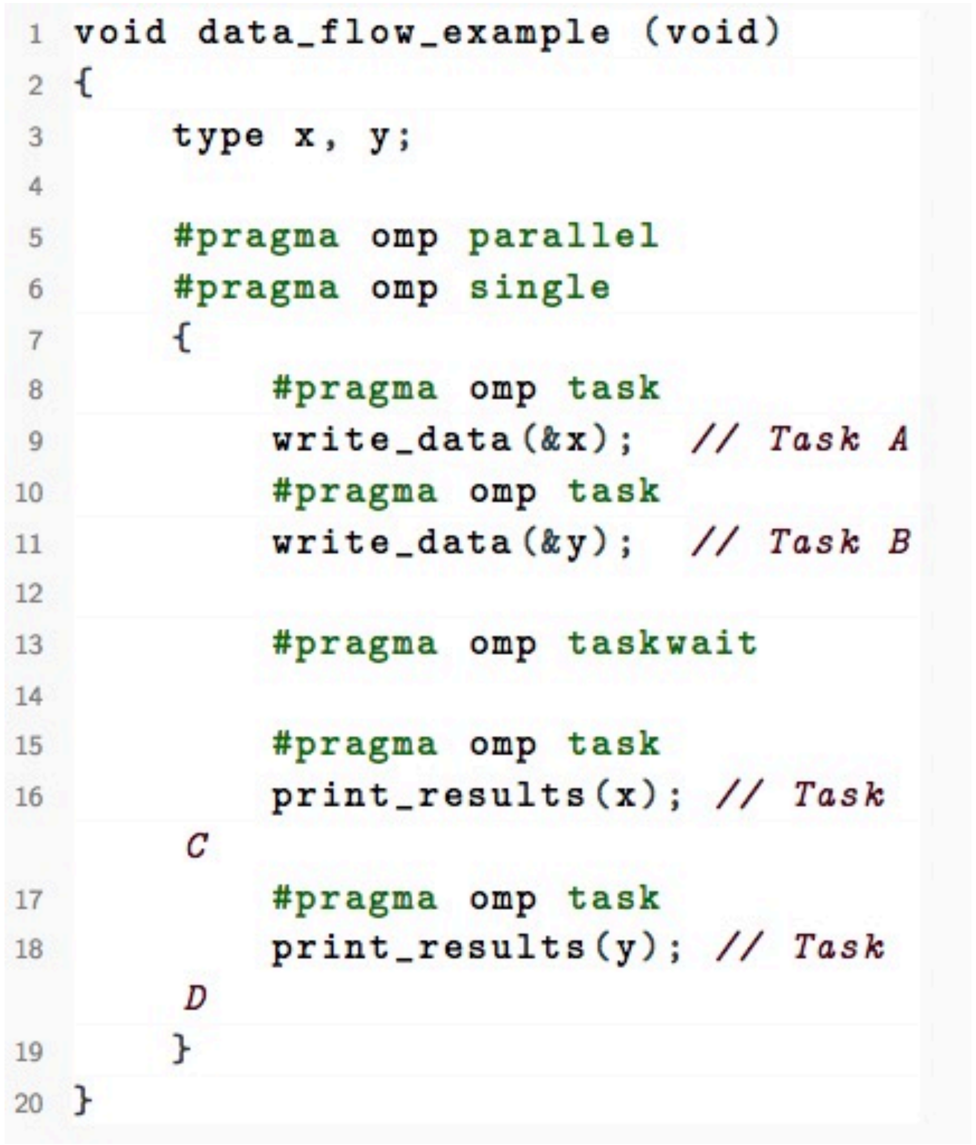
- Now that everything is working as intended, we would like to print is `fun to watch` at the end of the output string.
- This example illustrates the **asynchronous execution** of tasks.
- To fix this, you need to explicitly wait for the completion of the tasks with **taskwait** before printing "is fun to watch"

Program output:

```
$ OMP_NUM_THREADS=2 ./task-5
$ A race car is fun to watch
$ OMP_NUM_THREADS=2 ./task-5
$ A car race is fun to watch
```

What About Tasks with Dependencies on Other Tasks?

- Here, task A is writing some data that will be processed by task C. The same goes for task B and task D.
- The taskwait construct here makes sure task C won’t execute before task A and task D before task B.
- As a side effect, task C won’t execute until the execution of task B is over, creating some kind of **fake dependency** between task B and C.



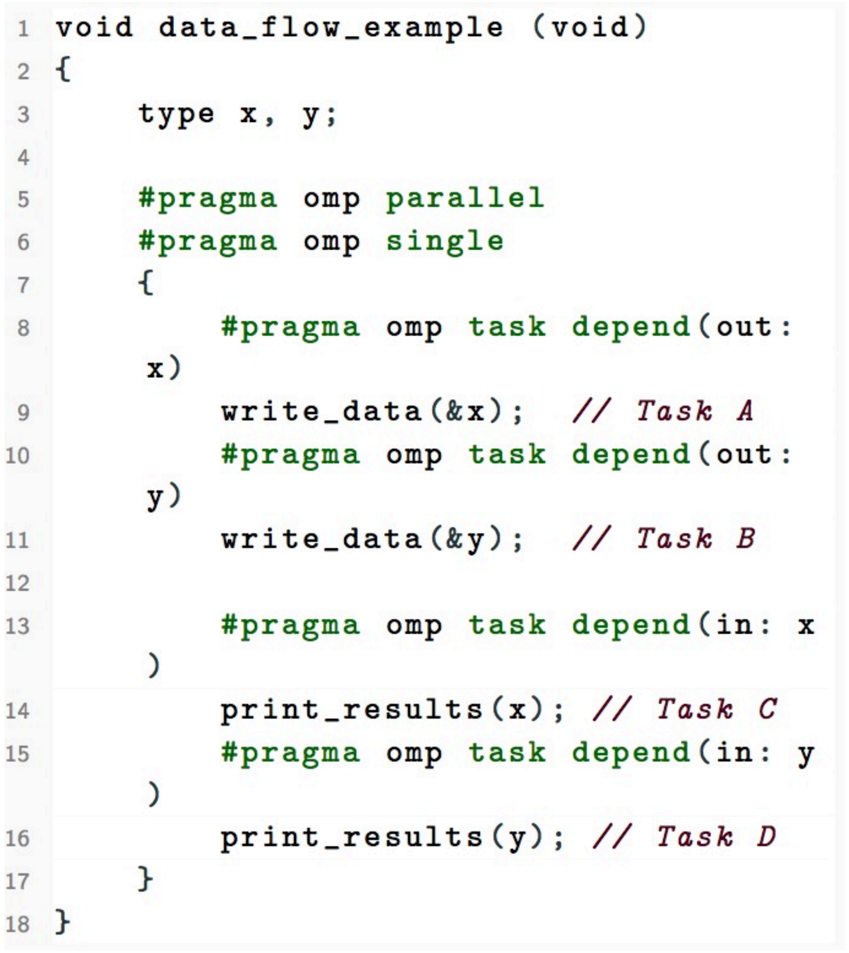
```
void data_flow_example (void)
{
    type x, y;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        write_data(&x);   // Task A
        #pragma omp task
        write_data(&y);   // Task B

        #pragma omp taskwait

        #pragma omp task
        print_results(x); // Task
    C
        #pragma omp task
        print_results(y); // Task
    D
    }
}
```

OpenMP Tasks Dependencies: Rationale

- The **depend** clause allows you to provide information on the way a task will access data.
- It is followed by an access mode that can be **in**, **out** or **inout**.
- Here are some examples of use for the **depend** clause:
  - **depend(in: x, y, z):** the task will read variables **x**, **y** and **z**
  - **depend(out: res):** the task will write variable **res**, any previous value of **res** will be ignored and overwritten
  - **depend(inout: k, buffer[0:n]):** the task will both read and write variable **k** and the content of **n** elements of **buffer** starting from index **0**

- The OpenMP runtime system dynamically decides whether a task is ready for execution or not considering its dependencies (there is no need for further user intervention here)

OpenMP Tasks Dependencies : Some Trivial Example



```
void data_flow_example (void)
{
    type x, y;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task depend(out: x)
        write_data(&x);   // Task A
        #pragma omp task depend(out: y)
        write_data(&y);   // Task B

        #pragma omp task depend(in: x )
        print_results(x); // Task C
        #pragma omp task depend(in: y )
        print_results(y); // Task D
    }
}
```

- Here is the previous example program written with tasks dependencies
- The **taskwait** construct is gone
  - The runtime system will rely on data dependencies to choose a ready task to execute
- In this version, task C could be executed before task B, as long as the execution of task A is over

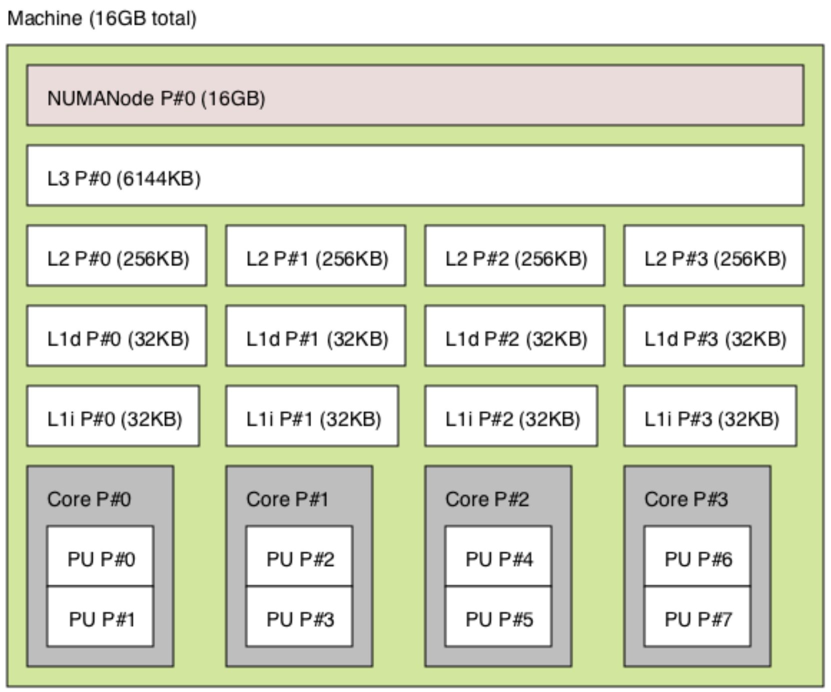
- Expressing dependencies sometimes helps unlocking more parallelism

Speeding up OpenMP applications

- **Preambule:** Have a closer look at your favorite/target platform
- Improving the execution of a parallel application **requires a good understanding of the target platform architecture**
- In particular, knowing about the following items is always useful:
  - The multicore processor: how many cores are available?
    - Which of them are physical/logical cores (HyperThreading and friends)?
  - The memory hierarchy: what kind of memory is available?
    - How is it organized?
  - The architecture topology: how (multicore) processors are connected together and how do they access memory?

Getting to Know Your Platform with hwloc

The hwloc library gathers valuable information about your platform and synthesize it into a generic representation.

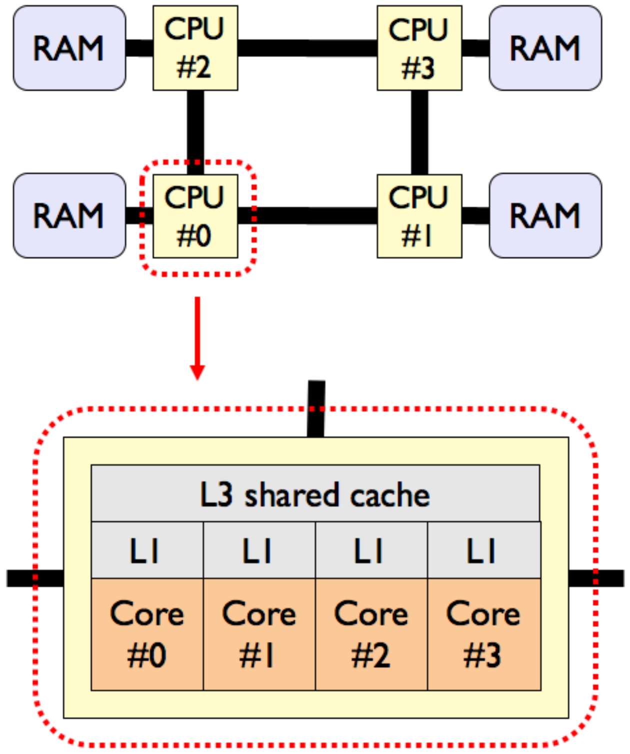


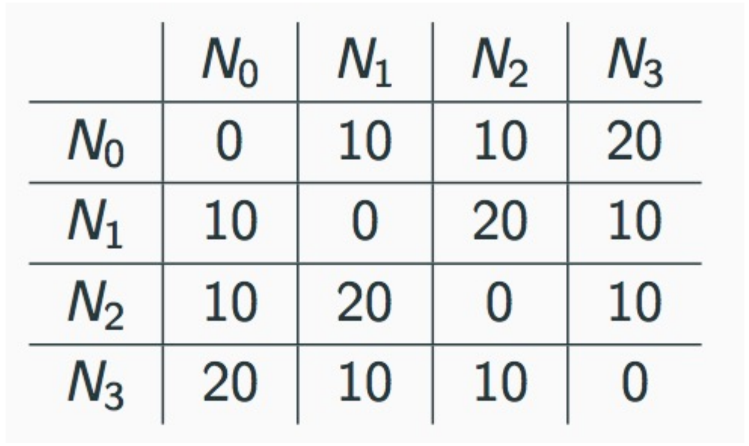
- Provides information about
  - the processing units (logical/physical cores)
  - the cache hierarchy
  - the memory hierarchy (NUMA nodes)
  - However, **hwloc does not provide the entire architecture topology** (the way processors are connected together).

https://www.open-mpi.org/projects/hwloc/

Understanding the Architecture Topology

- The operating system knows about the way processors are connected together to some extent. It can provide a distance table that roughly represent how many crossbars you need to cross to access a specific NUMA node (see the hwloc-distances program)



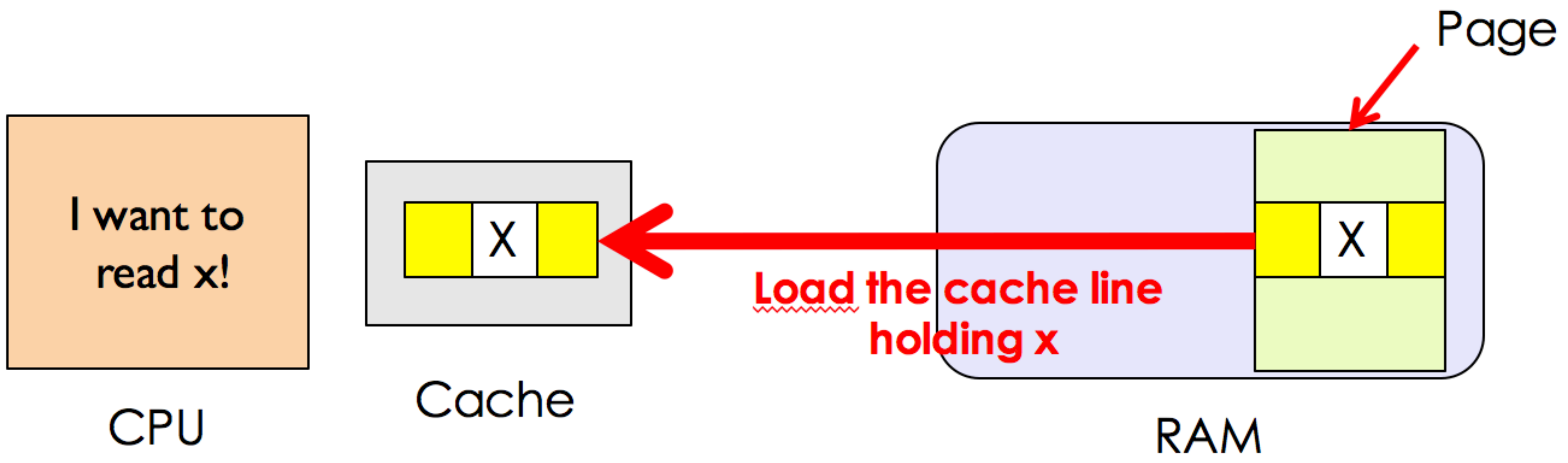


	N_0	N_1	N_2	N_3
N_0	0	10	10	20
N_1	10	0	20	10
N_2	10	20	0	10
N_3	20	10	10	0

A 4-nodes NUMA machine with the corresponding NUMA distance table

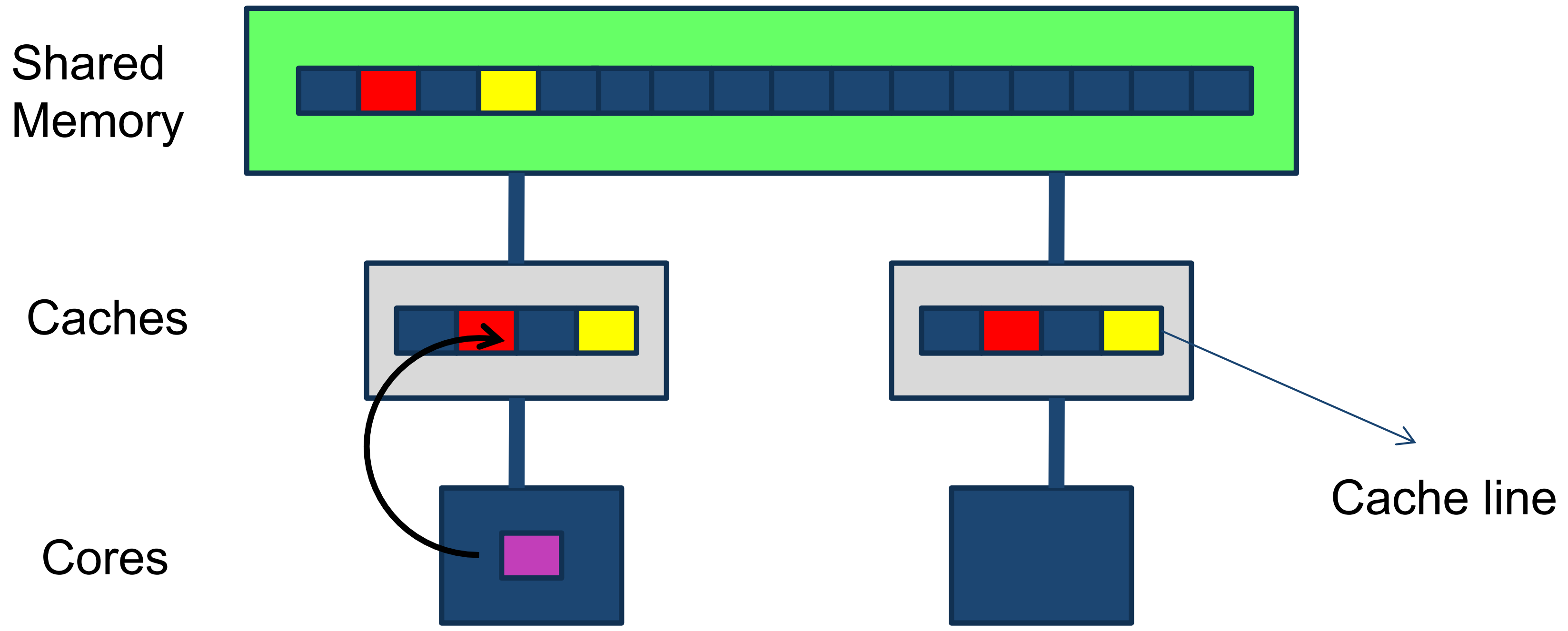
Cache Memory: Basic Concept

- A **cache** can be seen as a table of **cache lines** holding a predefined amount of memory (64B on most processors)
- Accessing a variable results in a **cache hit** if the corresponding cache line has already been cached (**fast memory access**)
- It can also result in a **cache miss** if the corresponding cache line is not cached yet. The hardware has to load the cache line to the cache before the processor can access it (**longer memory access**).



Performance: False Sharing effect

Cache coherency and the negative effect of false sharing can have a big impact on performance



Charging a line of a shared cache invalids the other copies of this line

Performance: False Sharing effect, contd.

Using data structures in memory may lead to a decrease of performance and a lack of scalability

- To get performance, use the cache
- If several cores manipulate different data items close in memory, the update of individual elements leads a load of a line of cache (to keep coherency with the main memory)

False sharing leads to bad performance when the following conditions are met

- Shared data a modified on # cores
- Several threads on # cores update data which are located on the same cache line
- These update appeards simultaneously and frequently

Performance: False Sharing effect, contd.

When data are only read, we don’t get false sharing

It can be avoided (or reduced) by

- Privatizing variables
- Increasing the array size or by using “padding”
- By increasing the packet size (modifying the way loop iterations are shared between threads)

Performance: False Sharing effect, contd.

```
Integer, dimension(n) :: a
…
!$OMP PARALLEL DO SHARED(nthreads,a) SCHEDULE(static,1)
            DO i=0, nthreads-1
                        a(i) = i
    END DO
!$OMP END PARALLEL DO
```

Nthreads : # threads executing the loop
If we suppose that every thread possess a copy of a in his local cache
Packet size of 1 leads to a false sharing phenomenon for each update

If a cache line can contain **C** elements of vector **a**, we can sole the problem by extending artificially the array dimensions (array padding)
We declare an array **a(C,n)** and remplace **a(i)** by **a(1,i)**

Naive Square Matrix Multiplication Algorithm

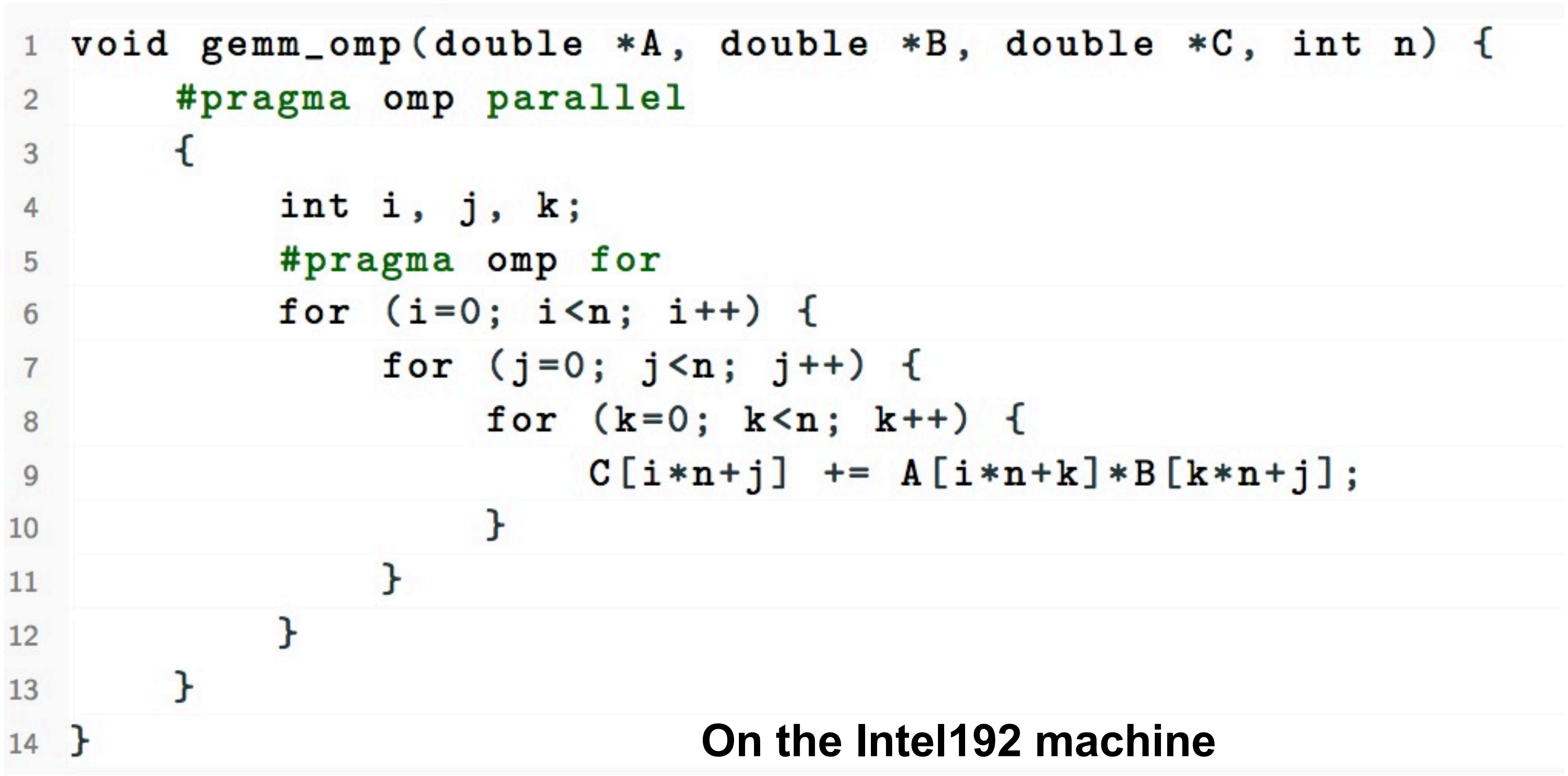
We consider a simple matrix multiplication algorithm involving square matrices of double precision floats.

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

where $C_0 = A_0B_0 + A_1B_4 + A_2B_8 + A_3B_{12}$

Let's implement this using what we've learned about OpenMP!

Speeding-Up OpenMP: Benefit from Cache Memory (2)



```
void gemm_omp(double *A, double *B, double *C, int n) {
    #pragma omp parallel
    {
        int i, j, k;
        #pragma omp for
        for (i=0; i<n; i++) {
            for (j=0; j<n; j++) {
                for (k=0; k<n; k++) {
                    C[i*n+j] += A[i*n+k]*B[k*n+j];
                }
            }
        }
    }
}
```

On the Intel192 machine

Serial time : 40.3018250s
Parallel time : 0.270773s
Achieved speed-up: 148

We Can Do Better : It's All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of C?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

$$\begin{aligned} C_0 &= A_0B_0 \\ &+ A_1B_4 \\ &+ A_2B_8 \\ &+ A_3B_{12} \end{aligned}$$

We Can Do Better : It's All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of C?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

$$\begin{aligned} C_0 &= A_0 B_0 \\ &+ A_1 B_4 \\ &+ A_2 B_8 \\ &+ A_3 B_{12} \end{aligned}$$

We Can Do Better : It’s All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of C?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

$$C_0 = A_0B_0$$
$$+ A_1B_4$$
$$+ A_2B_8$$
$$+ A_3B_{12}$$

We Can Do Better : It's All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of C?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

$$\mathrm{C}_0 = \mathrm{A}_0\mathrm{B}_0$$
$$+ \mathrm{A}_1\mathrm{B}_4$$
$$+ \mathrm{A}_2\mathrm{B}_8$$
$$+ \mathrm{A}_3\mathrm{B}_{12}$$

We Can Do Better : It's All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of C?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

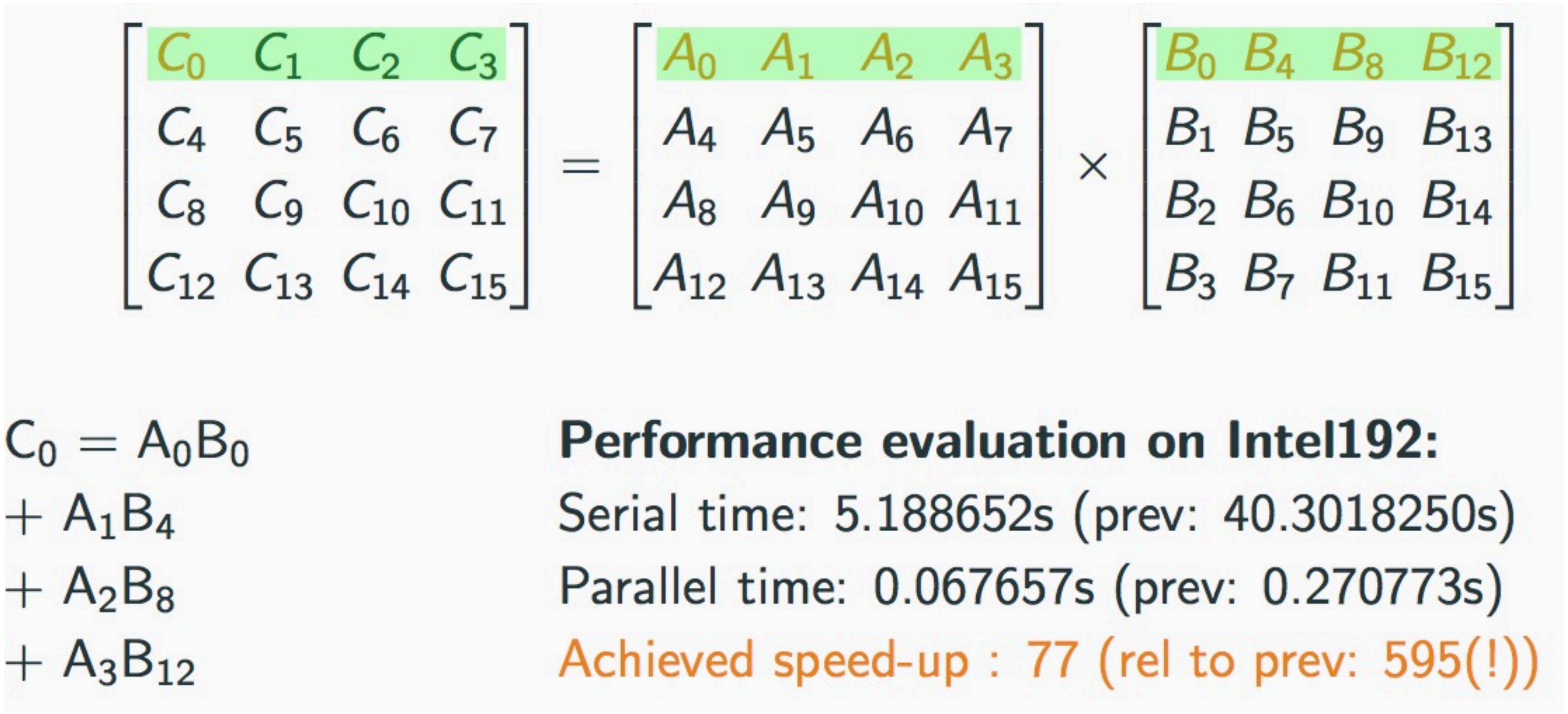
$$\begin{aligned} C_0 &= A_0B_0 \\ &+ A_1B_4 \\ &+ A_2B_8 \\ &+ A_3B_{12} \end{aligned}$$

Conclusion:

- Every access to A_i use the same cache line => cache friendly
- Every access to B_j use a different cache line => poor cache utilization

Deal with the B_j Situation

To improve cache utilization, we can transpose matrix B to make sure B_0, B_4, B_8 and B_{12} are stored on the same cache line



$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{bmatrix}$$

$$\begin{aligned} C_0 &= A_0B_0 \\ &+ A_1B_4 \\ &+ A_2B_8 \\ &+ A_3B_{12} \end{aligned}$$

Performance evaluation on Intel192:

Serial time: 5.188652s (prev: 40.3018250s)

Parallel time: 0.067657s (prev: 0.270773s)

Achieved speed-up : 77 (rel to prev: 595(!))