University of Minho
Informatics Department

Parallel Computing Paradigms (UC CPD)

OpenMP 3.0

João Luís Sobral
Bruno Medeiros

Why to use parallelism?

- Solve bigger problems
- Solve the same problem faster
- Architectures that demand it
  - Multi/Many-core
  - GPUs
- Energy consumption
- Maximize investment

- **Types of parallelism**
  - Programs = algorithms + data structures
  - **Data**– apply the same algorithm to different data sets in parallel
  - **Functional/task** – apply different parts of the algorithm to data
  - **Pipeline** - apply different parts of the algorithm to to different data sets (mix of previous)

Serial to Parallel Approach

1. Develop sequential code
2. Profile the code to identify most time-consuming parts
3. Migrate from sequential to parallel implementation

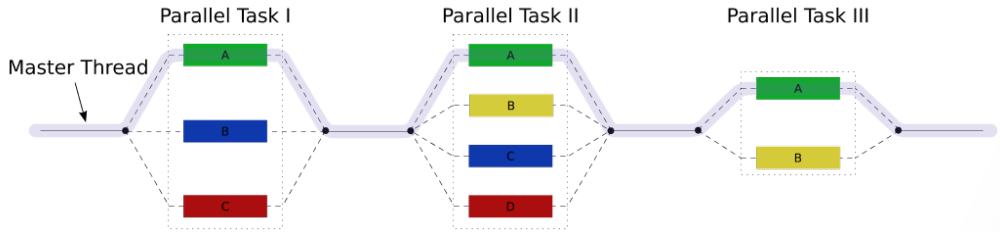
- **Not the best process**, just the easiest
  - Best parallel implementation may require a new design

- The OpenMP (most used) approach
  - Write a sequential code in a common language (e.g. C)
  - Add directives to parallelise portions of the code
  - Get a parallel program that computes the same result (serial to parallel equivalence)!

  - Programs have a mix of sequential and parallel parts
    - Parallel parts are defined with directives

Introduction to OpenMP

- OpenMP is a standard to Shared Memory (SM) parallel programming (e.g., on multi-core machines).
  - Based on: Compiler directives, Library routines and Environment variables;
  - Supports C/C++ and Fortran programming languages.
- Execution model is based on the **fork-join** model of parallel execution.



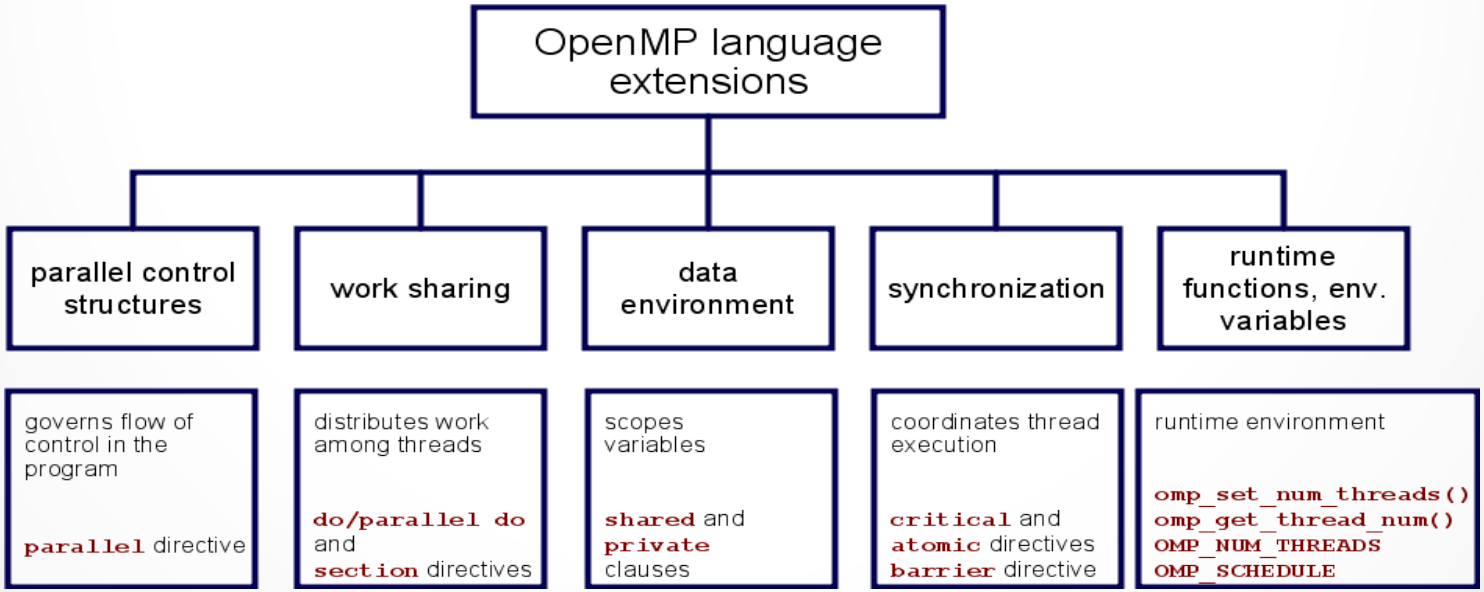
- Parallelism is specified through directives, added by the programmer to the code
  - the compiler implements the parallelism

OpenMP considerations:

- It is the programmer's responsibility to ensure **correctness** and **efficiency** of parallel programs
  - OpenMP itself **does not** solve problems as :
    - Data races, starvation, deadlock or poor load balancing (among others).
    - But, offers routines to solve problems like:
      - Load balancing or memory consistency.
- The creation/managing of threads is delegated to the compiler & OpenMP runtime:
  - + Easier to parallelize applications;
  - - Less control over the threads behaviour.
- By default, the number of parallel activities is defined at run-time according to available resources
  - e.g. 2 cores -> 2 threads
  - HT capability counts as additional cores

OpenMP: Programming Model

- An OpenMP program begins with a single thread (**master thread**).
- **Parallel regions** create a team of parallel activities;
- **Work-sharing** constructs generate work for the team to process;
- **Data sharing** clauses specify how variables are shared within a parallel region;



Overview of OpenMP constructs (1)

- OpenMP directives format for C/C++ applications:

  #pragma omp *directive-name [clause[[,] clause]...] new-line*
  block of code // group of statements separated by semicolons, enclosed in braces

- **Parallel Construct**
  - *#pragma omp parallel* Creates a team of threads.
- **Work-sharing Constructs**
  - *#pragma omp for* Assignment of loop iterations to threads.
  - #pragma omp sections Assignment of blocks of code (section) to threads.
  - #pragma omp single Restricts a code of block to be executed by a single thread.

Overview of OpenMP constructs (2)

- **Tasking Constructs**
  - #pragma omp task Creation of a pool of tasks to be executed by threads.

- **Master & Synchronization Constructs**
  - #pragma omp master Restricts a block of code to be executed only the master thread.
  - #pragma omp critical Restricts the execution of a block of code to a single thread at a time.
  - #pragma omp barrier Makes all threads in a team to wait for the remaining.
  - #pragma omp taskwait wait for the completion of the current task child's.
  - #pragma omp atomic Ensures that a specific storage location is managed atomically.
  - #pragma omp flush Makes a thread's temporary view of memory consistent with memory.
  - #pragma omp ordered Specifies a block of code in a loop region that will be executed in the order of the loop iterations.

Parallel Region

- When a thread encounters a parallel construct, a team of threads is created (**FORK**);
- The thread which encounters the parallel region becomes the **master** of the new team;
- **All threads** in the team (including the master) execute the region;
- At end of parallel region, all threads synchronize, and join master thread (**JOIN**).

Parallel region syntax

```
#pragma omp parallel [clauses]
{
        code_block
}
```

Where clause can be:

`if` (*scalar-expression*)

`num_threads` (*integer-expression*)

`private` (*list*)

`firstprivate` (*list*)

`shared` (*list*)

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

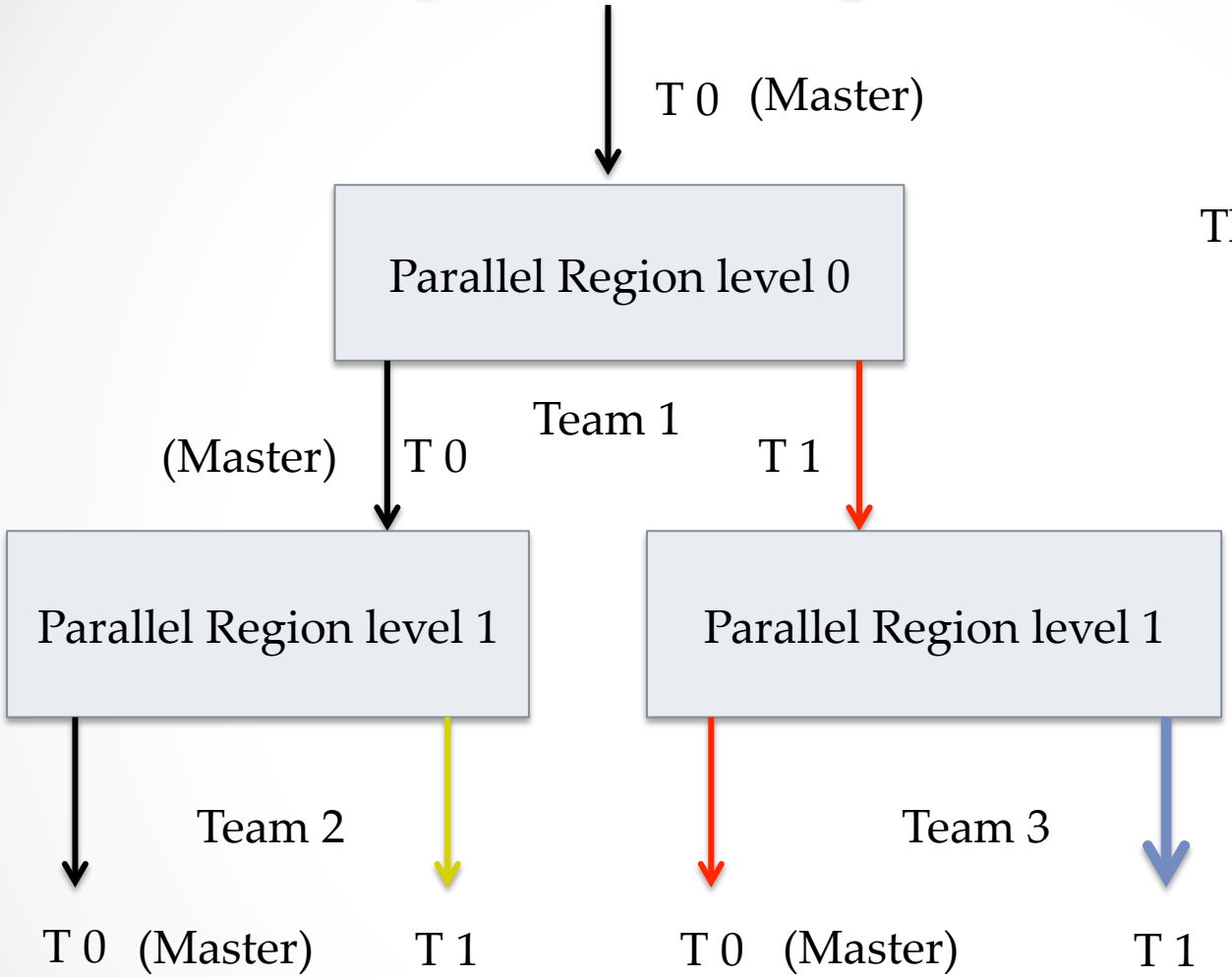
Nested Parallel Region

- If a thread in a team executing a parallel region encounters another parallel directive, it creates a new team, and becomes the master of this team;

- If **nested parallelism** is disabled, then no additional team of threads will be created.

- To enable/disabled -> **omp_set_nested(x);**

Nested region example



Thread marked with red is slave on team 1 but master on team 2.

Loop Construct

- The *for* loop iterations are distributed across threads **in the team;**
  - The distribution is based on:
    - **chunk_size**, by default is 1;
    - **schedule** by default is static.
- Loop schedule:
  - **Static –** Iterations divided into chunks of size ***chunk_size*** assigned to the threads in a team in a **round-robin** fashion;
  - **Dynamic –** the chunks are assigned to threads in the team as the threads request them;
  - **Guided** - similar to dynamic but the chunk size decreases during execution.
  - **Auto –** the selection of the scheduling strategy is delegated to the OpenMP implementation.

Parallel region syntax

```
#pragma omp for[clauses]
{
        code_block
}
```

Where clause can be:

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
schedule(kindf, chunk_size])
collapse(n)
ordered
nowait
```

Loop Constructors

- schedule(**static**) vs schedule(**dynamic**)
  - **Static** has lower overhead;
  - **Dynamic** has a better load balance approach;
  - Increasing the chuck size in the dynamic for:
    - Diminishing of the scheduling overhead;
    - Increasing the possibility of load balancing problems.

- Lets consider the following loop that we want to parallelize using 2 threads, being void f(int i) a given function

```
#pragma omp parallel for schedule ( ?)
 for(I = 0; I < 100; I++)
    f(i);
```

What is the most appropriated type of scheduling?

Parallel for with ordered clause

- #pragma omp for schedule(static) ordered

```
for (i = 0; i < N; ++i)
 {
    … // do something here (in parallel)
    #pragma omp ordered
    {
     printf("test() iteration %d\n", i);
    }
 }
```

Parallel execution of code sections

- Supports heterogeneous tasks:

```
#pragma omp parallel
{
   #pragma omp sections
    {
      #pragma omp section
      {
                taskA();
      }
      #pragma omp section
      {
                taskB();
      }
      #pragma omp section
      {
                taskC();
      }
   }
}
```

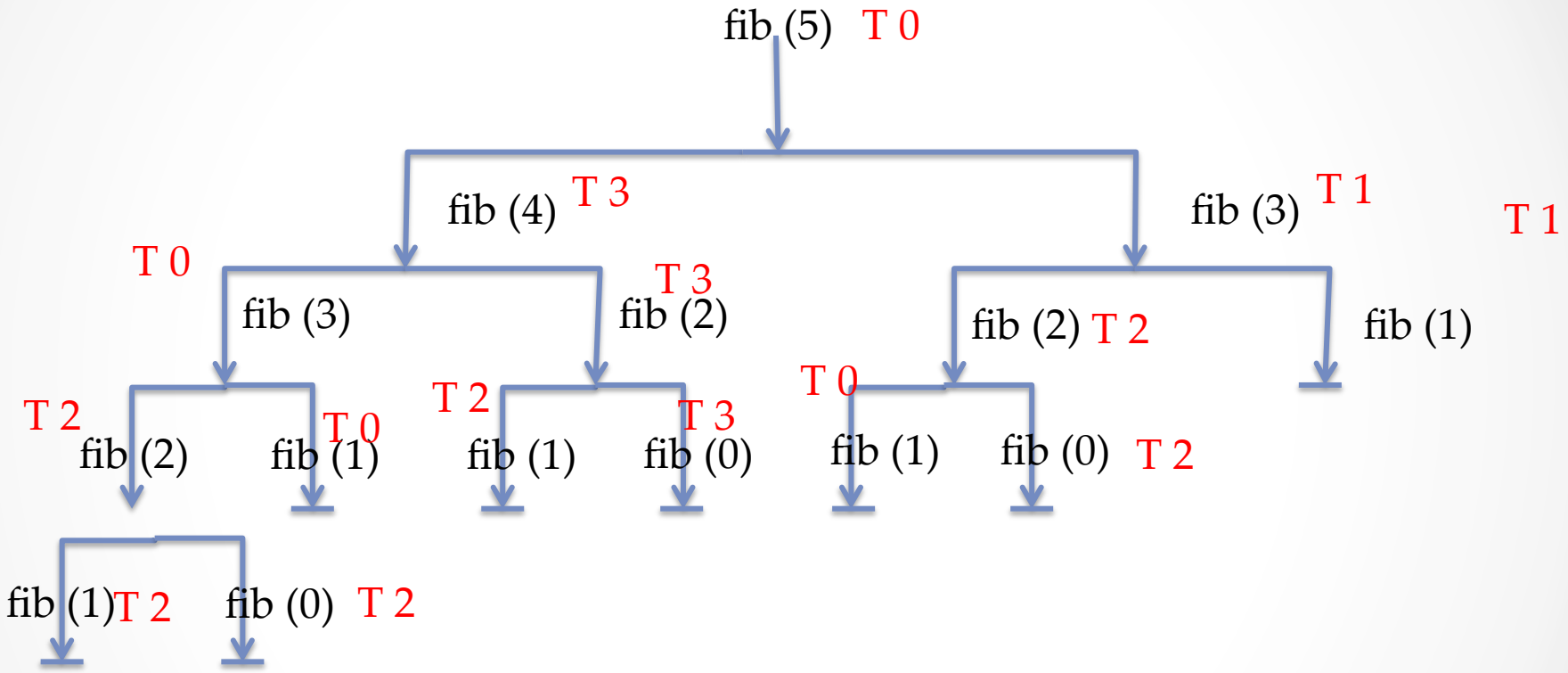
- The section blocks are divided among threads in the team;
- Each section is executed only once by threads in the team.
- There is an implicit barrier at the end of the section construct unless a nowait clause is specified
- Allow the following clauses:
  - private (list);
  - firstprivate(list);
  - lastprivate(list);
  - reduction(operator:list)

Task constructor:

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

       #pragma omp task shared(j) firstprivate(n)
       j=fib(n-2);

       #pragma omp taskwait
       return i+j;
    }
}

int main()
{
  int n = 10;

  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

- When a thread encounters a task construct, a task is generated;
- Thread can immediately execute the task, or can be executed latter on by any thread on the team;
- OpenMP creates a pool of tasks to be executed by the active threads in the team;
- The **taskwait** directive ensures that the tasks generated are completed before the return statements.
- Although, only one thread executes the **single** directive and hence the call to fib(n), **all four threads** will participate in executing the tasks generated.

Execution Tree Exemplified



Synchronization Constructs:

- Critical regions (executed in mutual exclusion):
  - #pragma omp critical [name]

    updateParticles();
  - Restricts the execution of the associated structured blocks to a single thread at a time;
  - Works inter-teams (i.e., global lock)
  - An optional name may be used to identify the critical construct.

- Atomic Operations (fine-grain synchronization):
  - #pragma omp atomic

    A[i] += x;
  - The memory in will be updated atomically. It does not make the entire statement atomic; only the memory update is atomic.
  - A compiler might use special hardware instructions for **better** performance than when using ***critical***.

Avoid/reduce synchronisation

- Reduction of multiple values (in parallel):

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int i = 0;  i<100; i++) {
            sum += array[i];
}
```

- Thread reuse across parallel regions

```
# pragma omp parallel {
#pragma omp for
for(int i = 0;  i<100; i++)
        …
#pragma omp for
    for(int j= 0; j<100; j++)
        …
}
```

Data Sharing

- What happens to variables in parallel regions?
  - Variables declared **inside** are **local** to each thread;
  - Variables declared **outside** are **shared**
- Data sharing clauses:
  - **private(varlist)** => each variable in varlist becomes private to each thread, initial values not specified.
  - **firstprivate(varlist) =>** Same as private, but variables are initalized with the value outside the region.
  - **lastprivate(varlist) =>** same as private, but the final value is the last loop iteration's value.
  - **reduction (op:var)** => same as lastprivate, but the final value is the result of reduction of private values using the operator "**op**".
- Directives for data sharing:
  - #pragma omp **threadlocal** => each thread gets a local copy of the value.
  - **copyin** clause copies the values from thread master to the others threads.

Environment variables

- **OMP_SCHEDULE**
  - sets the *run-sched-var* ICV for the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types (i.e., **static**, **dynamic**, **guided**, and **auto**).
- **OMP_NUM_THREADS**
  - sets the *nthreads-var* ICV for the number of threads to use for **parallel** regions.
- **OMP_DYNAMIC**
  - sets the *dyn-var* ICV for the dynamic adjustment of threads to use for **parallel** regions.
- **OMP_NESTED**
  - sets the *nest-var* ICV to enable or to disable nested parallelism.
- **OMP_STACKSIZE**
  - sets the *stacksize-var* ICV that specifies the size of the stack for threads created by the OpenMP implementation.
- **OMP_WAIT_POLICY**
  - sets the *wait-policy-var* ICV that controls the desired behavior of waiting threads.
- **OMP_MAX_ACTIVE_LEVELS**
  - sets the *max-active-levels-var* ICV that controls the maximum number of nested active parallel regions.
- **OMP_THREAD_LIMIT**
  - sets the *thread-limit-var* ICV that controls the maximum number of threads participating in the OpenMP program.

OpenMP Rotines

- omp_set_num_threads / omp_get_num_threads
- omp_get_max_threads
- omp_get_thread_num.
- omp_get_num_procs.
- omp_in_parallel.
- omp_set_dynamic / omp_get_dynamic.
- omp_set_nested / omp_get_nested.
- omp_set_schedule / omp_get_schedule
- omp_get_thread_limit.
- omp_set_max_active_levels / omp_get_max_active_levels
- omp_get_level.
- omp_get_ancestor_thread_num.
- omp_get_team_size.
- omp_get_active_level

- **Locks**
  - void omp_init_lock(omp_lock_t *lock);
  - void omp_destroy_lock(omp_lock_t *lock);
  - void omp_set_lock(omp_lock_t *lock);
  - void omp_unset_lock(omp_lock_t *lock);
  - int omp_test_lock(omp_lock_t *lock);
- **Timers**
  - double omp_get_wtime(void);
  - double omp_get_wtick(void);

OpenMP versions and compiler support

Version	Main new features	Compiler support		
		GCC	ICC	Clang
2.5 (May 2005)		4.2		
3.0 (May 2008)	- Task / taskwait	4.4	11.0	
3.1 (July 2011)	- Final/mergeable - Taskyield - Min/max reductions in C++ - OMP_PROC_BIND	4.7	12.1	3.7 (p)
4.0 (July 2013)	- Cancel - Declare reduction - SIMD - Taskgroup - Device construct	4.9.0	15.0	
4.5 (Nov 2016)	- Taskloop - new ordered clauses - Offloading changes	6.1	17.0	3.9 (p)

OpenMP discussion

- Pros:
  - Portable
  - Simple (compared to MPI)
  - Incremental parallelism
  - Sequential semantics
  - (Limited) support for GPGPU
- Cons:
  - Requires a compiler with OpenMP support
  - Possible data races (and other thread synchronization problems)
  - Scalability
    - Limited to shared memory
    - No (efficient) support for distributed memory
  - Complex parallelism requires explicit parallel code

- References

- OpenMP3.1ReferenceManual

http://www.openmp.org/mp-documents/OpenMP3.1.pdf