Master Informatics Eng.

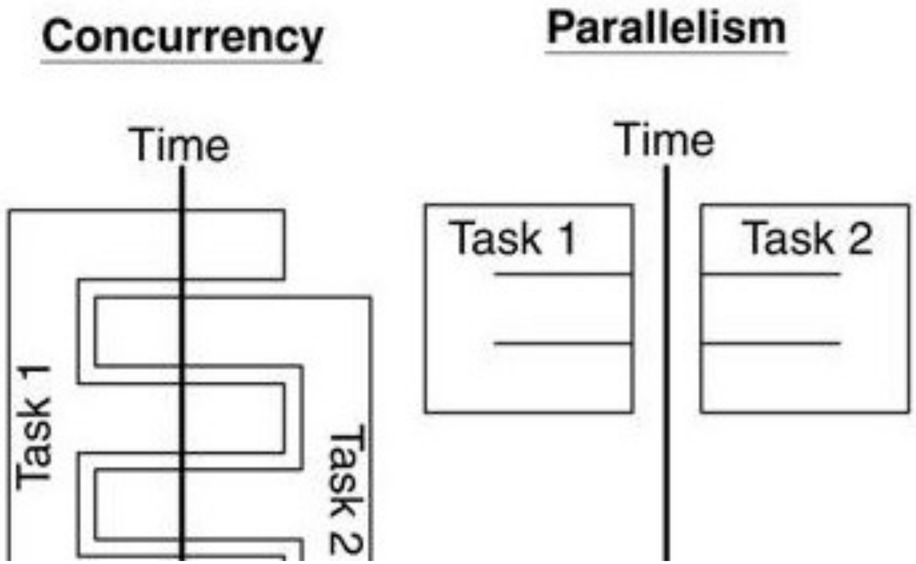
2021/22

A.J.Proença

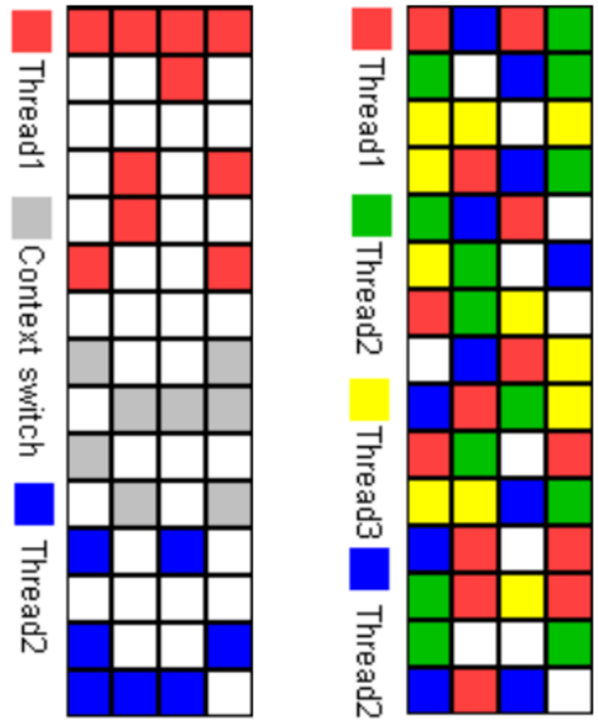
Programming in Shared Memory

(most slides are from previous year)

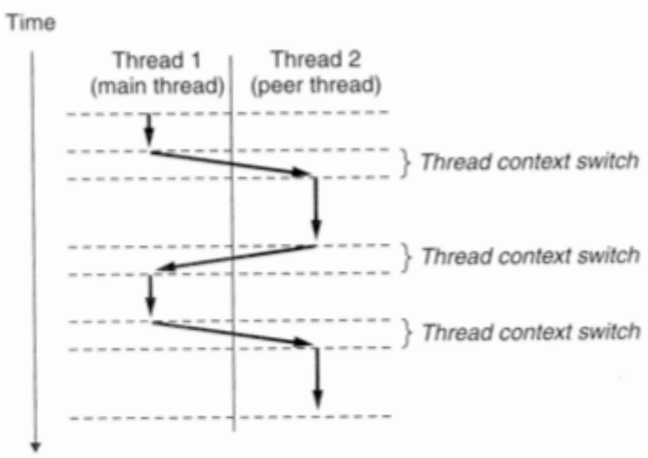
Logic *vs.* physical parallelism



Intel SMT



Scheduling



Specification of concurrency/parallelism

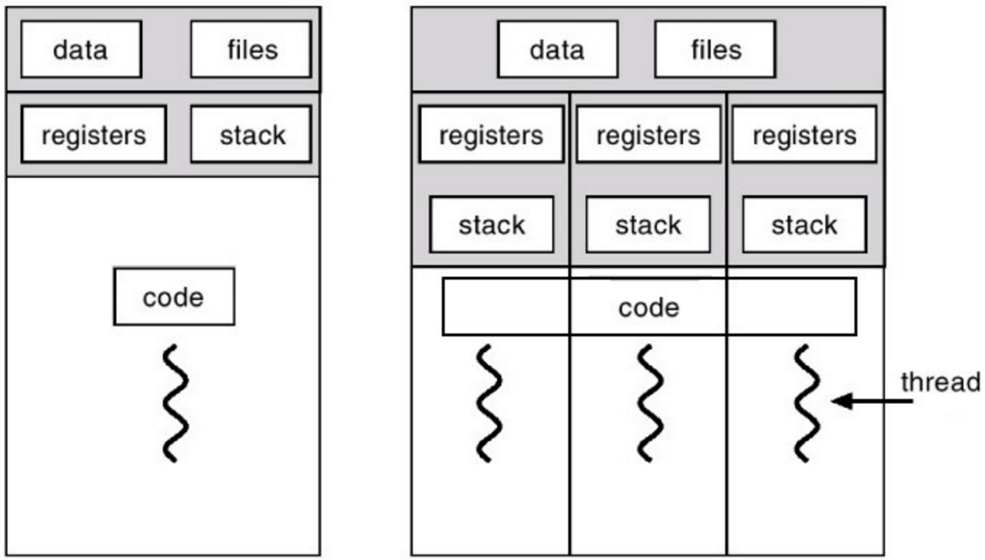
Processes

- Used for unrelated tasks
  - (e.g., a program)
- Own address space
  - Address space is proteded from other process
- Swithching at the kernel level

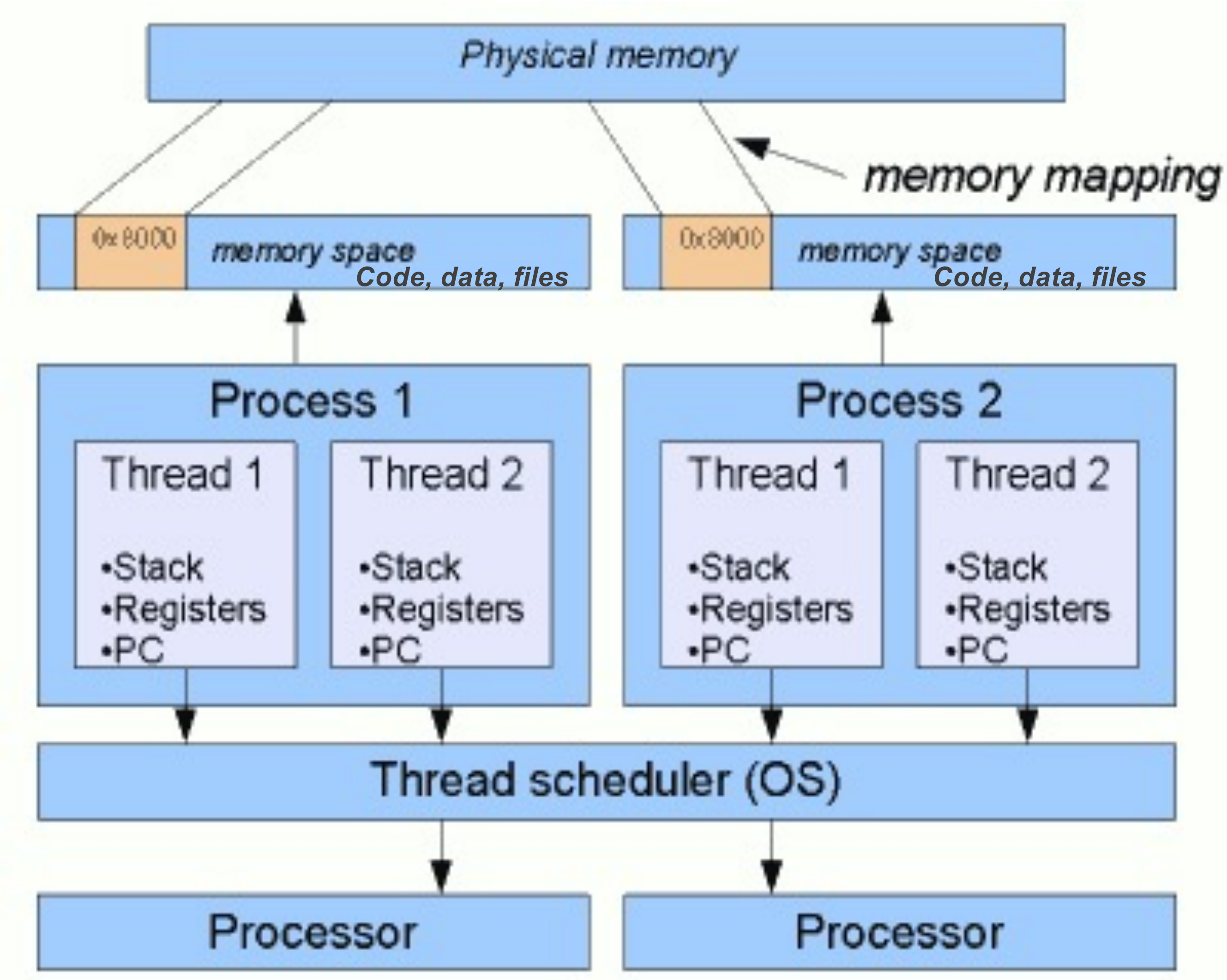
Threads

- Are part from the same job
- Share address space, code, data and files
- Swithching at the user or kernel level

Every process has at lest one thread



Threads *vs.* Processes

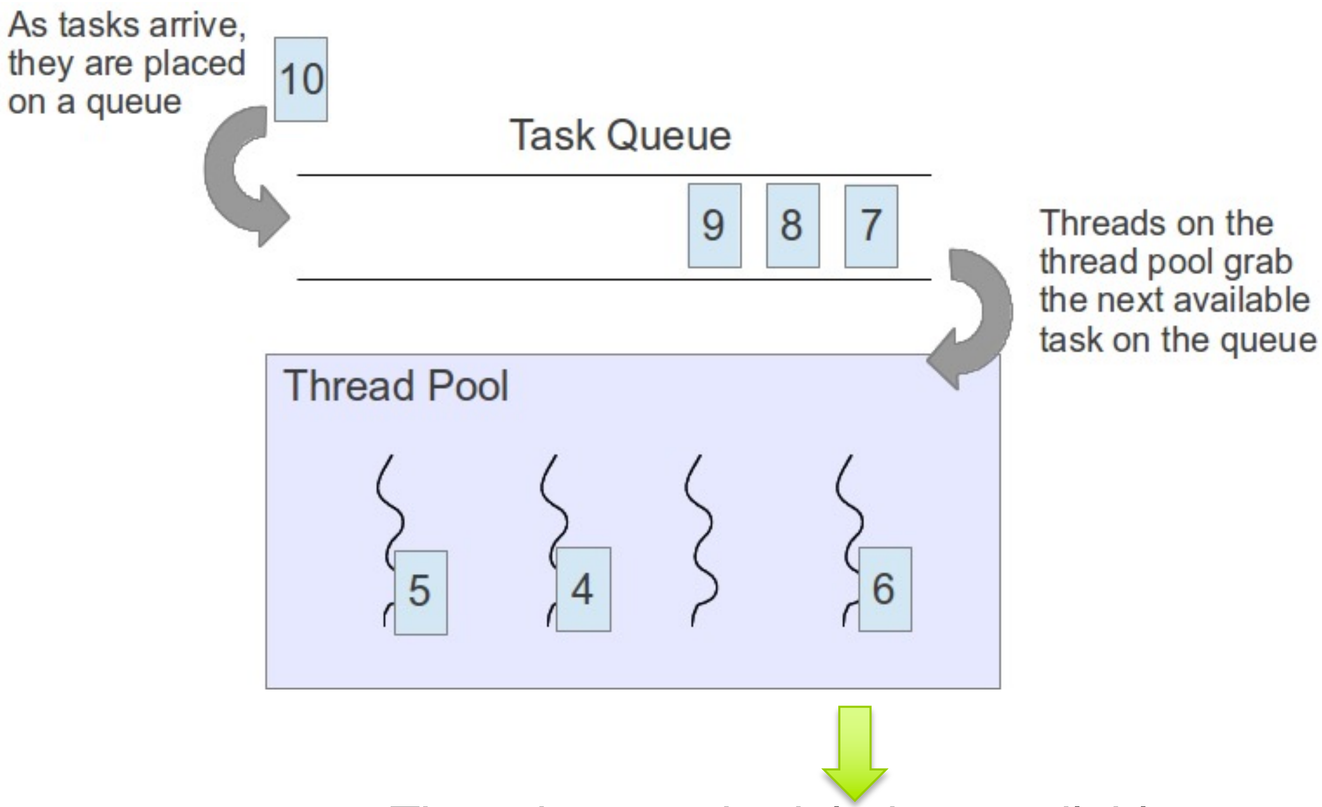


Processes/Threads *vs.* Tasks

- **Task**: sequence of instructions
- **Thread/process**: execution context for a task
- **Processor/core**: hardware that runs a thread/process

In Java

- Runnable object
- Thread
- Processor core



Threads are scheduled on available cores

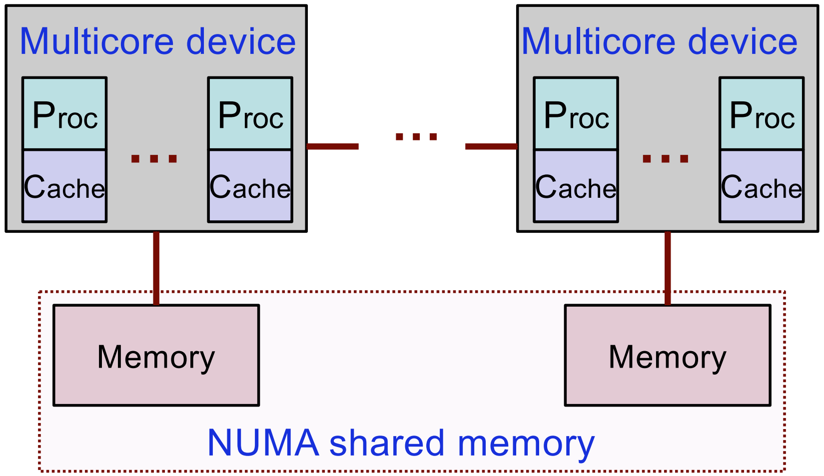
Desenvolvimento de Aplicações Paralelas

Partição do problema e dos dados a processar

- Identifica **oportunidades de paralelismo**:
  - Define um elevado número de **tarefas** (de grão fino)
  - Pode obter várias decomposições alternativas
- Duas vertentes complementares na identificação das **tarefas**:
  - **Decomposição dos dados -** identifica dados que podem ser processados em paralelo
    - enfoque nos **dados** a processar e na sua divisão em conjuntos que podem ser processados em paralelo
  - **Decomposição funcional –** identifica fases do algoritmo que podem ser efectuadas em paralelo
    - enfoque no **processamento** a realizar, dividindo este processamento em tarefas independentes
- A partição deve obter um número de **tarefas**, pelo menos, uma ordem de magnitude superior ao número de unidades de processamento
  - Introduz flexibilidade nas fases posteriores do desenvolvimento.
- Tarefas de dimensões idênticas facilitam a distribuição da carga
- O número de tarefas deve aumentar com a dimensão do problema.

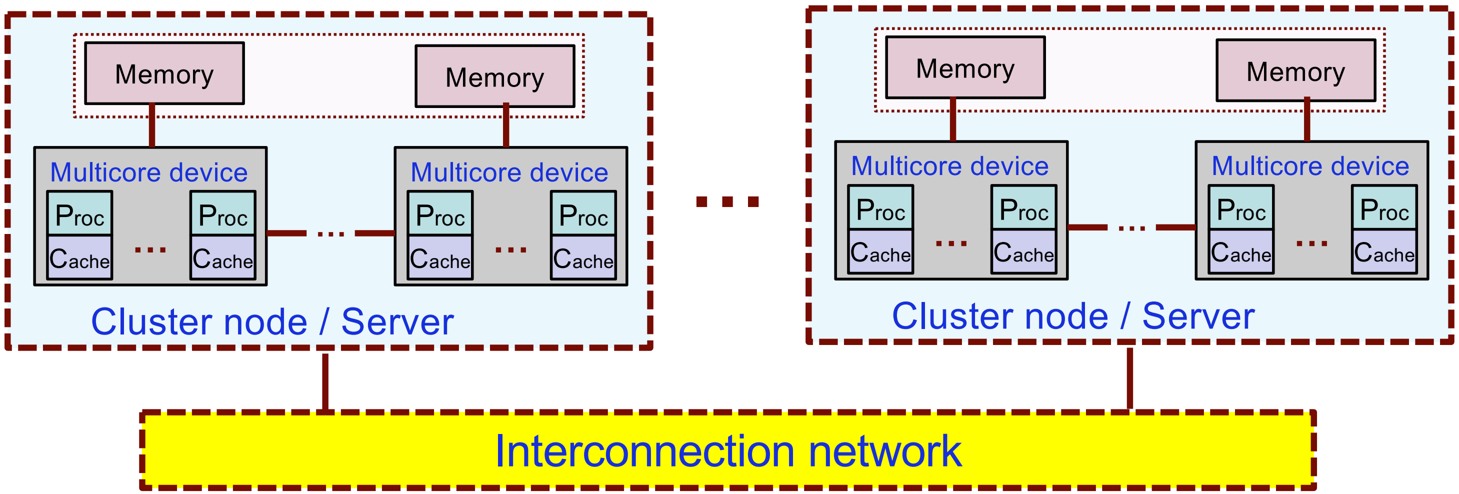
Explicit parallel computing *(1)*

- Current homogeneous parallel systems *(1)*
  - parallelism on single or multiple devices *(same motherboard)*
    - each core can be multithreaded
    - single physical mem addr space
    - paradigm: shared mem program
      - Cilk Plus (http://www.cilkplus.org/)
        extension to C & C++ to support
        data & task parallelism
      - **OpenMP** (http://openmp.org/wp/)
        C/C++ and Fortran directive-based parallelism



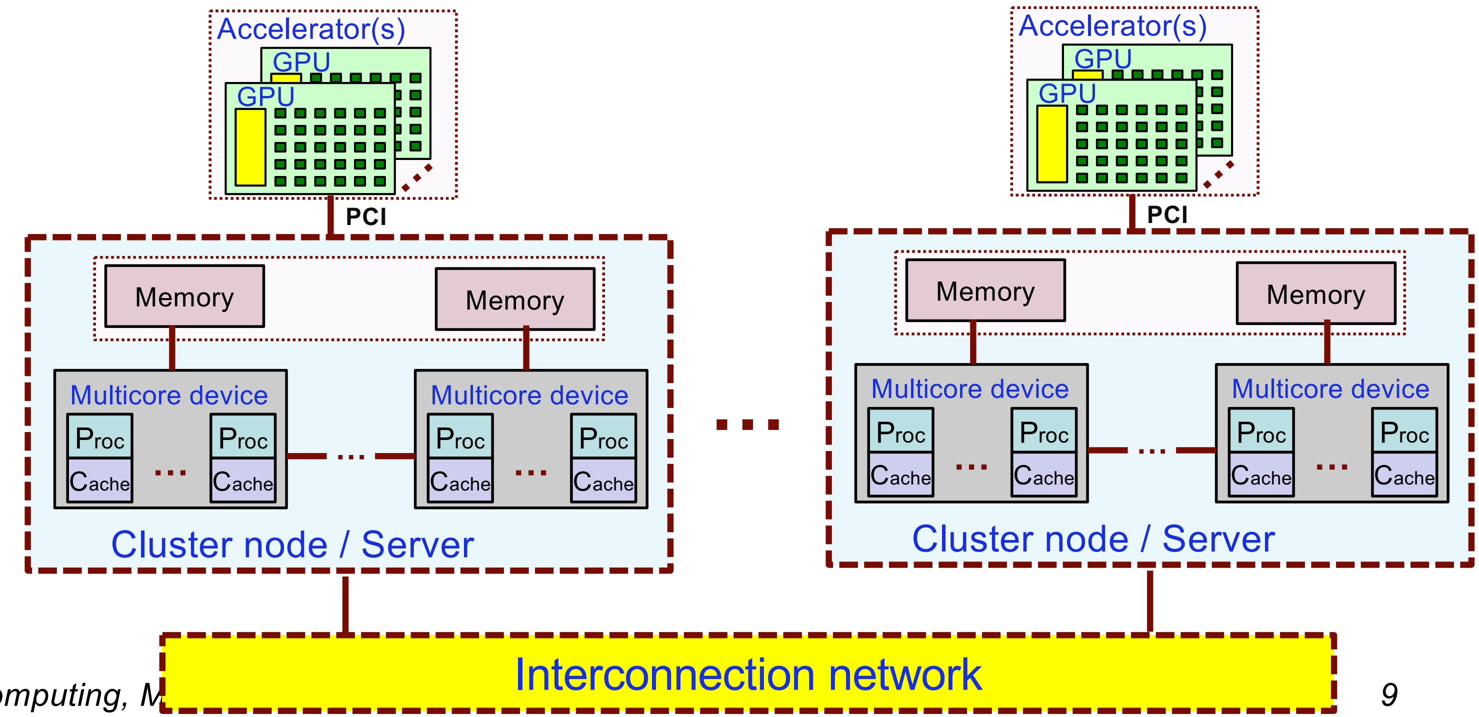
Explicit parallel computing *(2)*

- Current homogeneous parallel systems *(2)*
  - on multiple boards *(or multiple nodes/servers)*
    - each node with its private memory space
    - paradigm among nodes: distributed memory passing
      - MPI (https://en.wikipedia.org/wiki/Message_Passing_Interface) library for message communication on scalable parallel systems



***Explicit* parallel computing** *(3)*

- Common heterogeneous systems *(GPU as accelerator unit)*
  - application specific (GPU excellent for numerical computations)
  - key suppliers: **NVidia**, AMD, Intel, …
  - programming: OpenCL, **CUDA**, …



Parallel Programming Models

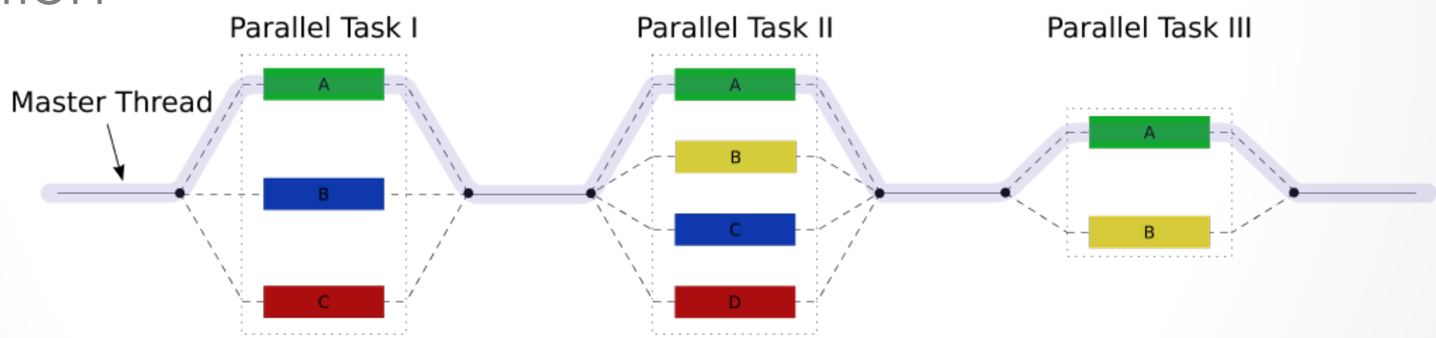
- Two general models of parallel program
  - Task parallel
    - problem is broken down into tasks to be performed
    - individual tasks are created and communicate to coordinate operations
  - Data parallel
    - problem is viewed as operations of parallel data
    - data distributed across processes and computed locally
- Characteristics of scalable parallel programs
  - Data domain decomposition to improve data locality
  - Communication and latency do not grow significantly

Shared Memory Parallel Programming

- Shared memory address space
- (Typically) easier to program
  - Implicit communication via (shared) data
  - Explicit synchronization to access data
- Programming methodology
  - Manual
    - multi-threading using standard thread libraries
  - Automatic
    - parallelizing compilers
    - OpenMP parallelism directives
  - Explicit threading (e.g. POSIX threads)

Introduction to OpenMP

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

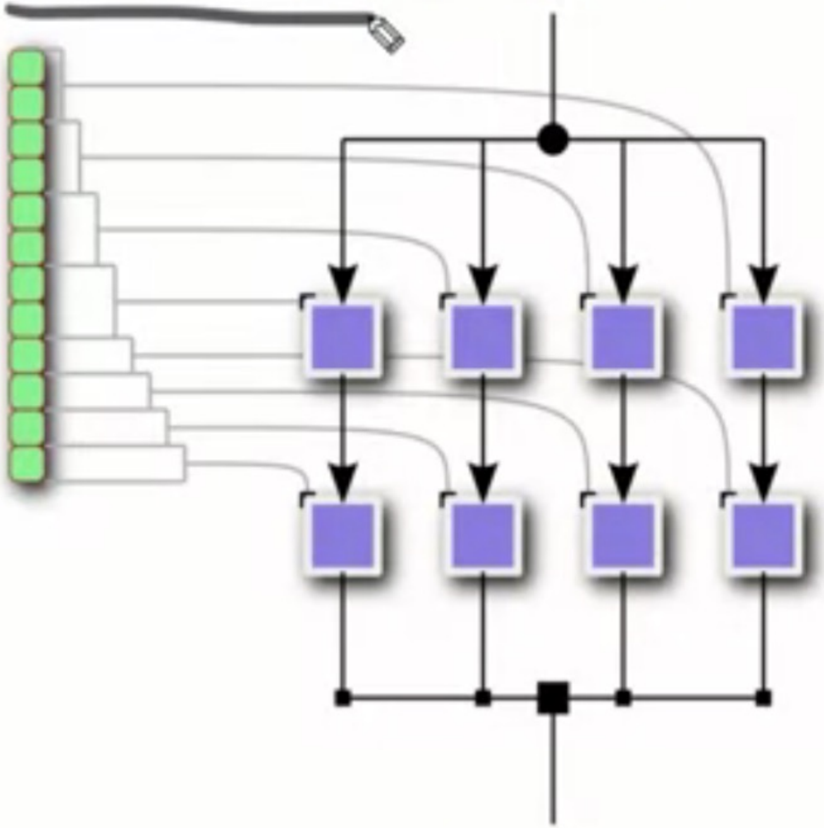


- Parallelism is specified through directives, added by the programmer to the code
  - o the compiler implements the parallelism

Parallel `for-loops` in OpenMP

Loop-Centric Parallelism: For-Loops in OpenMP

- Simultaneously launch multiple threads
- Scheduler assigns loop iterations to threads
- Each thread processes one iteration at a time



Parallelizing a `for`-loop.

https://www.coursera.org/learn/parallelism-ia

Directive for parallel loops in OpenMP

Loop-Centric Parallelism: For-Loops in OpenMP

The OpenMP library will distribute the iterations of the loop following the `#pragma omp parallel for` across threads.

```c
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  printf("Iteration %d is processed by thread %d\n",
         i, omp_get_thread_num());
  // ... iterations will be distributed across available threads...
}
```

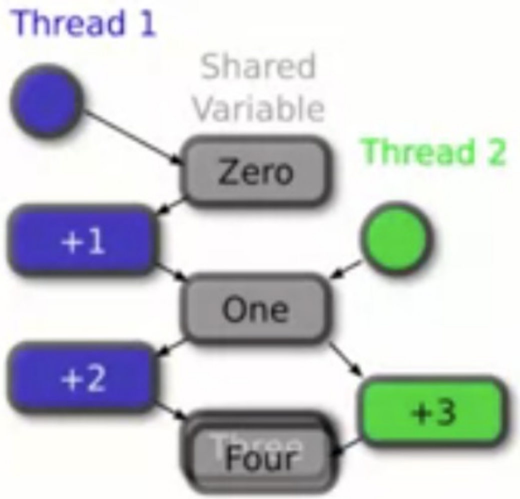
https://www.coursera.org/learn/parallelism-ia

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

Races conditions in multithreading

Race Conditions and Unpredictable Program Behavior



Race Condition!

- Occurs when 2 or more threads access the same memory address, and at least one of these accesses is for writing

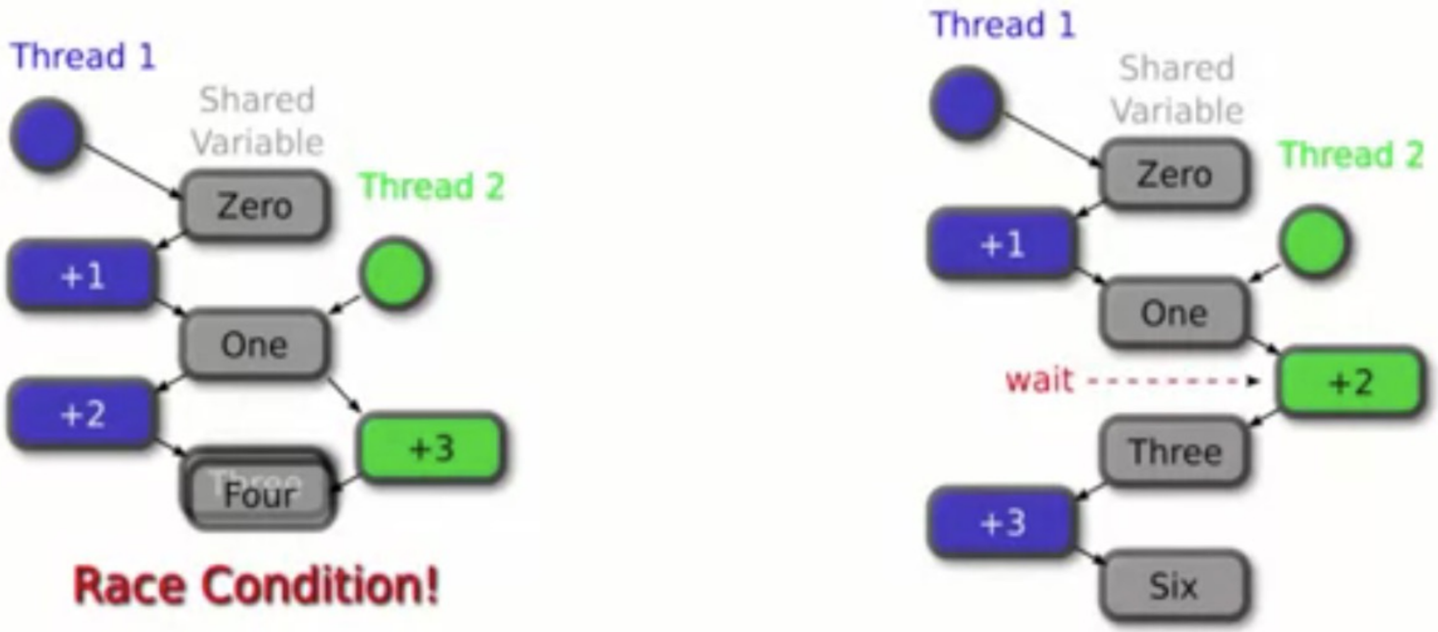
Directive in OpenMP

```
int total = 0;
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  // Race condition
  total = total + i;
}
```

https://www.coursera.org/learn/parallelism-ia

Avoiding data races with mutexes

Mutexes



- Mutual exclusion conditions (mutexes) protect data races by serializing code.

https://www.coursera.org/learn/parallelism-ia

Mutexes in OpenMP

Mutexes in OpenMP

```
#pragma omp parallel
  {
    // parallel code
#pragma omp critical
    {
      // protected code
      // multiple lines
      // many variables
    }
  }
```

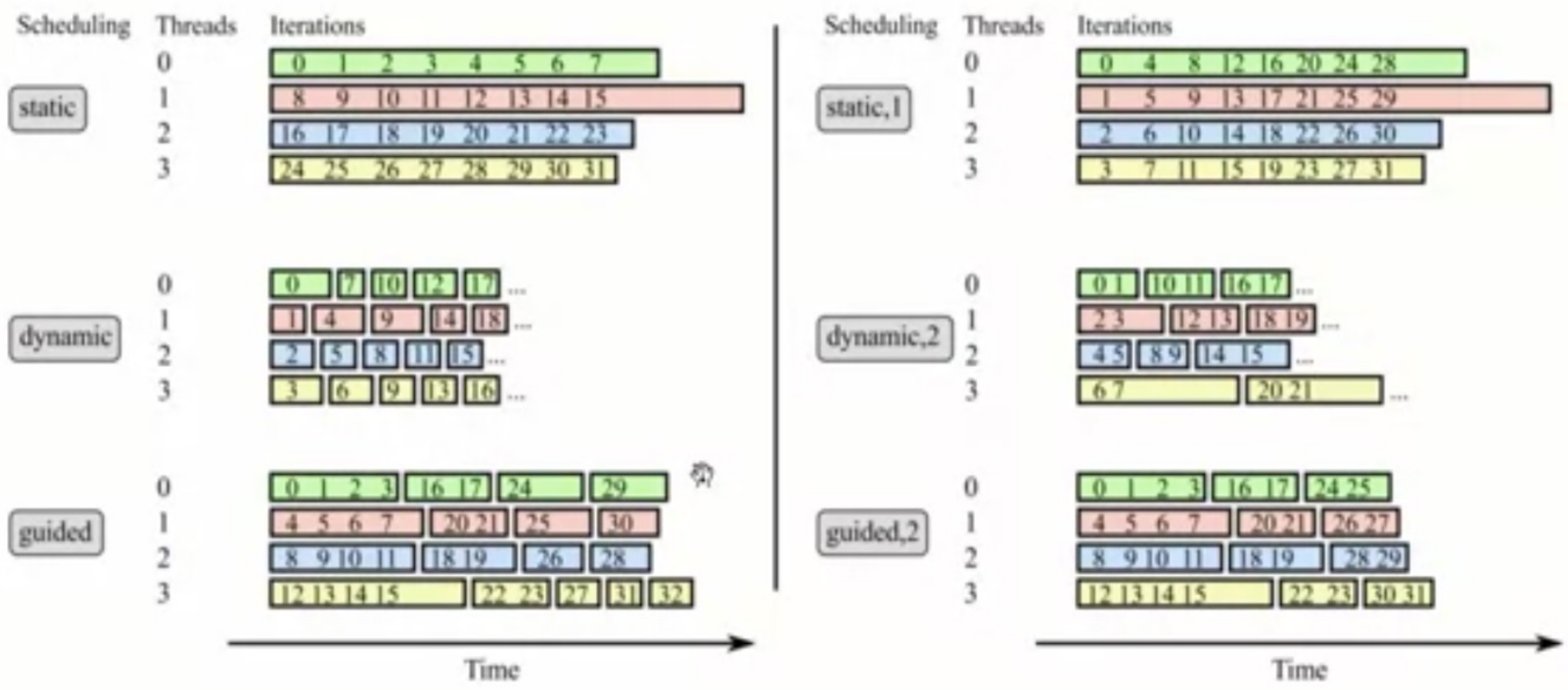
```
#pragma omp parallel
  {
    // parallel code
#pragma omp atomic
      // protected code
      // one line
      // specific operations
      // on scalars
      total += i;
  }
```

Good parallel codes minimize the use of mutexes.

https://www.coursera.org/learn/parallelism-ia

Load balancing loop iterations

Loop Scheduling Modes in OpenMP



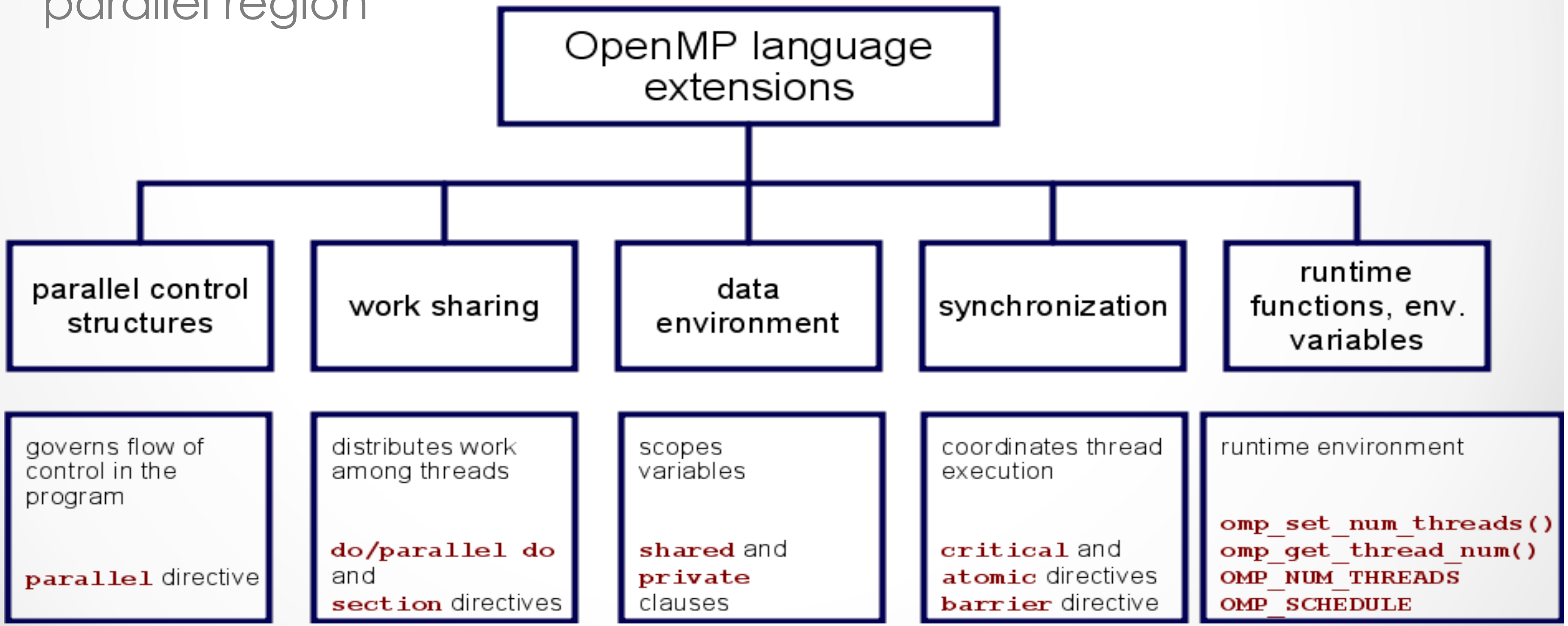
https://www.coursera.org/learn/parallelism-ia

COLFAX RESEARCH

intel

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*

- **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 (standard v2.5)**
  - #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 temporary view of mem 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)
```

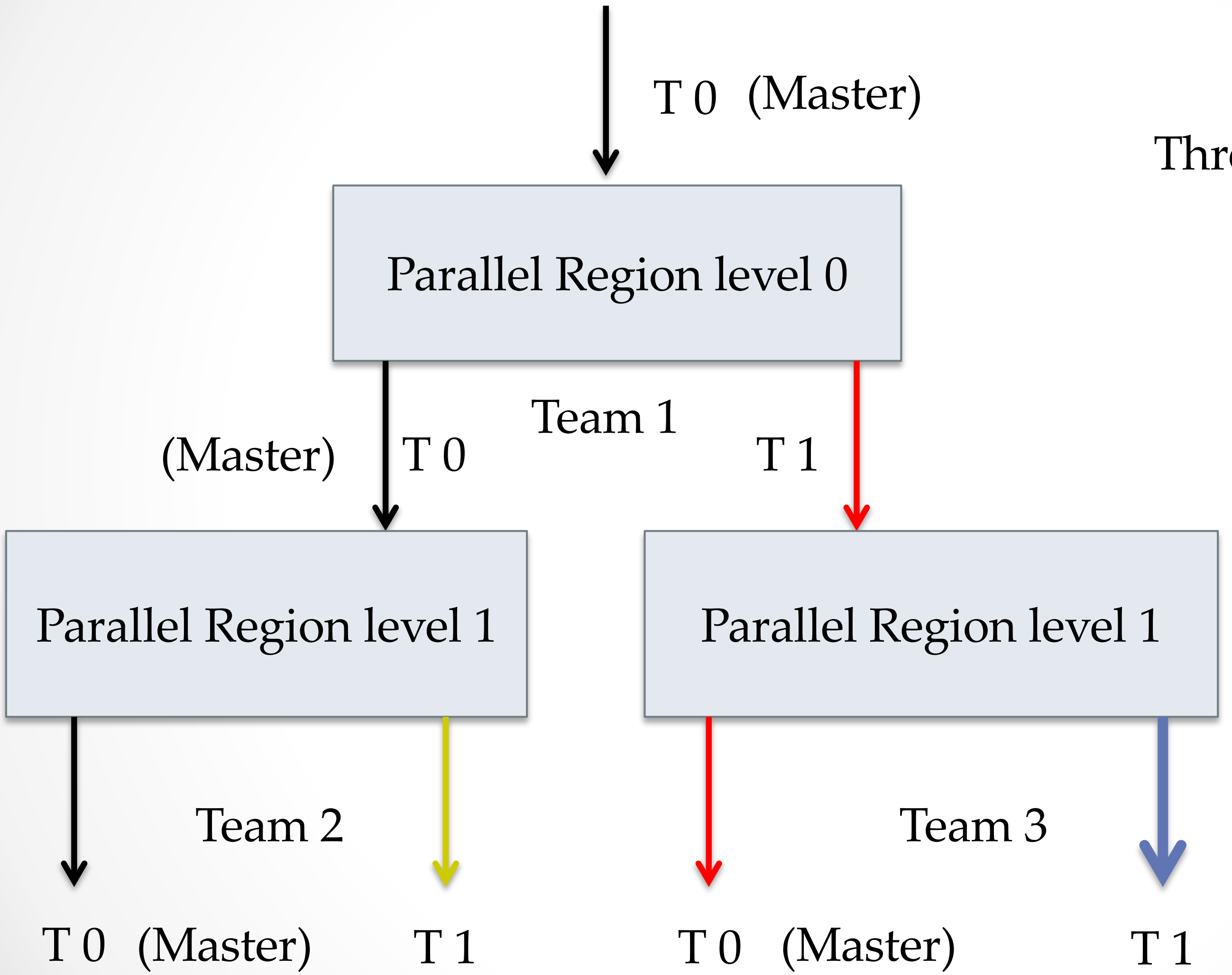
Nested Parallel Region

- 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(kind[, 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();
      }
   }
}
```

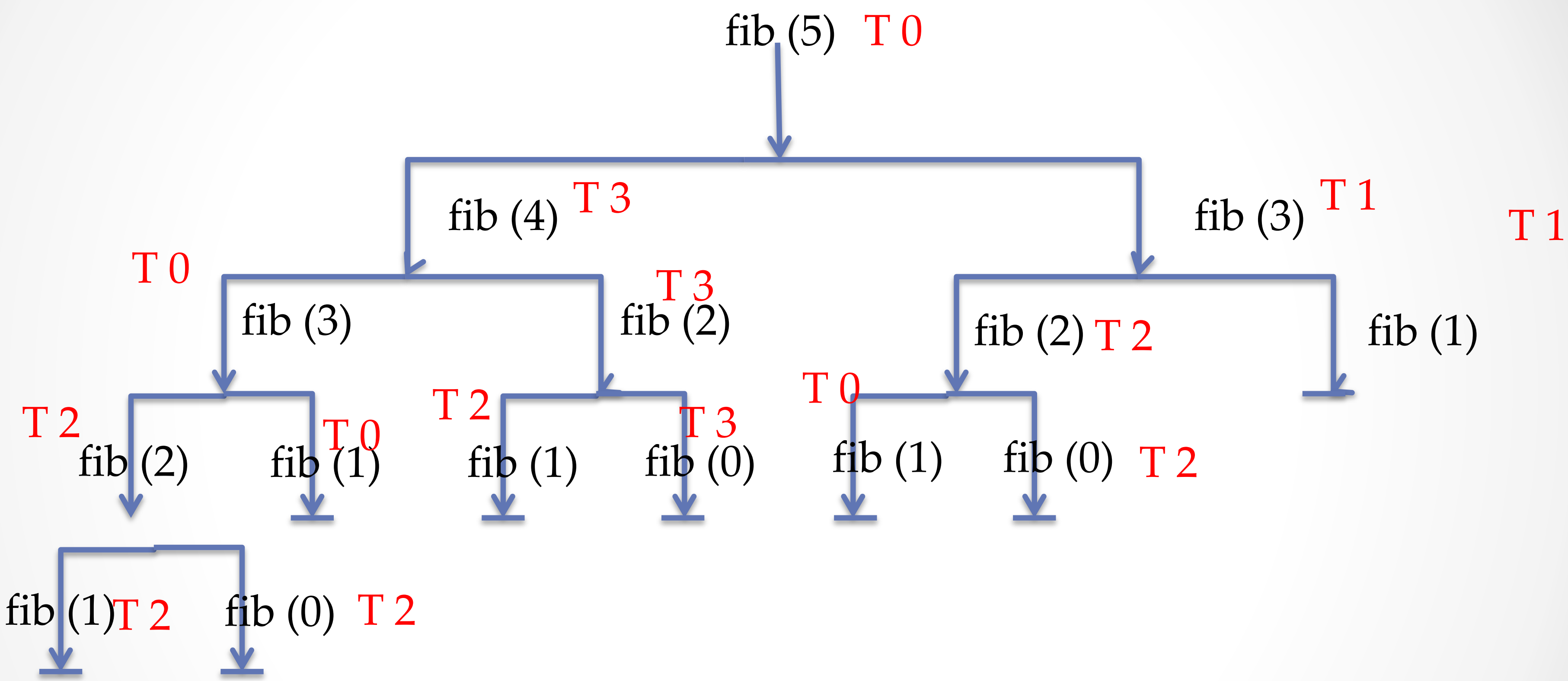
- The section blocks are divided among threads in the team;
- Each section is executed only once by threads in the team.
- There is an implicit barrier at the end of the section construct unless a nowait clause is specified
- Allow the following clauses:
  - private (list);
  - firstprivate(list);
 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

Task constructor (OpenMP 2.5):

```c
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



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.

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***.

Synchronization Constructs:

- Atomic Operations (fine-grain synchronization):

data-race.c 〉 No Selection

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

int main(){
    double result={0};

    #pragma omp parallel for shared(result)
    for(int i=0; i<1000000;i++) {
        result+=sin(i);
    }
    printf("%f",result);
}
```

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

int main(){
    double result={0};

    #pragma omp parallel for shared(result)
    for(int i=0; i<1000000;i++) {
        #pragma omp atomic
        result+=sin(i);
    }
    printf("%f",result);
}
```

```
    vmovsd  (%r12), %xmm1

L4:
    vxorpd  %xmm0, %xmm0, %xmm0
    vcvtsi2sd     %ebx, %xmm0, %xmm0
    vmovsd  %xmm1, 8(%rsp)
    addl    $1, %ebx
    call    _sin      ; return value in %xmm0
    vmovsd  8(%rsp), %xmm1  ; result in %xmm1
    cmpl    %ebx, %ebp
    vaddsd  %xmm0, %xmm1, %xmm1
    jne     L4

    vmovsd  %xmm1, (%r12)
```

```
L12:
    addl    $1, %ebx
    cmpl    %ebx, %r12d
    je      L9
L5:
    vxorpd  %xmm0, %xmm0, %xmm0
    vcvtsi2sd     %ebx, %xmm0, %xmm0
    call    _sin
    movq    0(%rbp), %rcx
    movq    (%rcx), %rdx
L4:
    vmovq   %rdx, %xmm2
    movq    %rdx, %rax
    vaddsd  %xmm2, %xmm0, %xmm1
    vmovq   %xmm1, %rsi
    lock cmpxchgq   %rsi, (%rcx)
    cmpq    %rax, %rdx
    je      L12
    movq    %rax, %rdx
    jmp     L4
```

Repeat until successful update

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++)
      …
}
```

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);