POLITECNICO DI MILANO

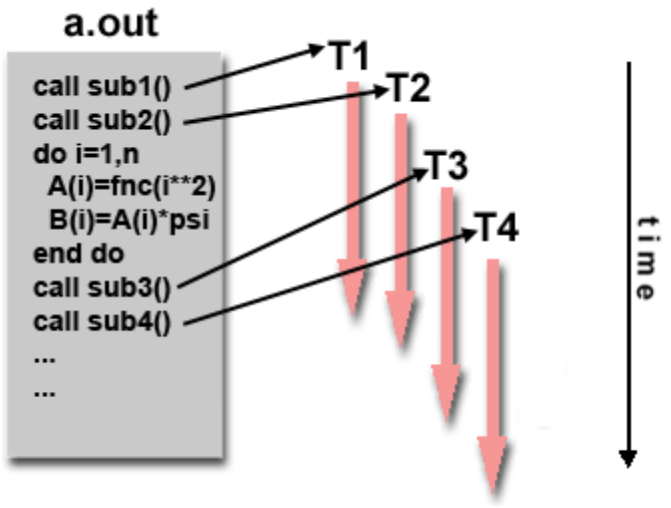
OpenMP part 1

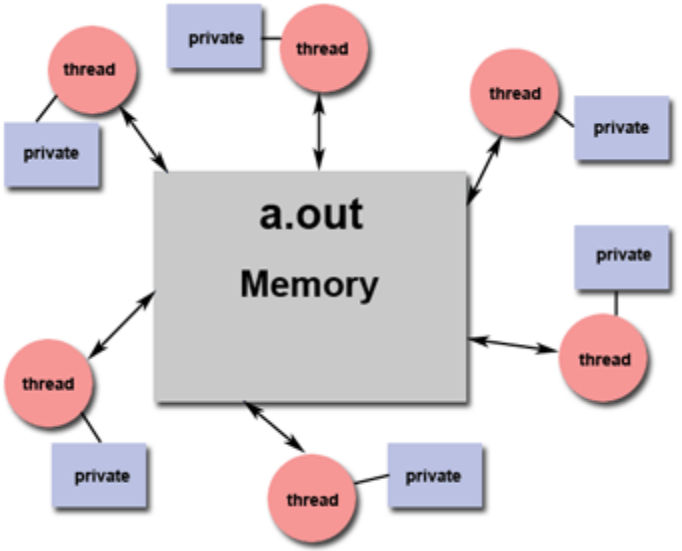
Parallel Computing

Serena Curzel
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
serena.curzel@polimi.it

Threads model

- Single process with multiple threads that can execute concurrently
- Each thread has local data, but it can access all resources acquired by the main process





Threads implementations

- The programmer is responsible for handling parallelism and synchronization, usually through
  - A library of subroutines
  - A set of compiler directives
- Historically, hardware vendors have implemented their own proprietary versions of threads
- We will see two different standards:
  - POSIX Threads (Pthreads)
  - OpenMP

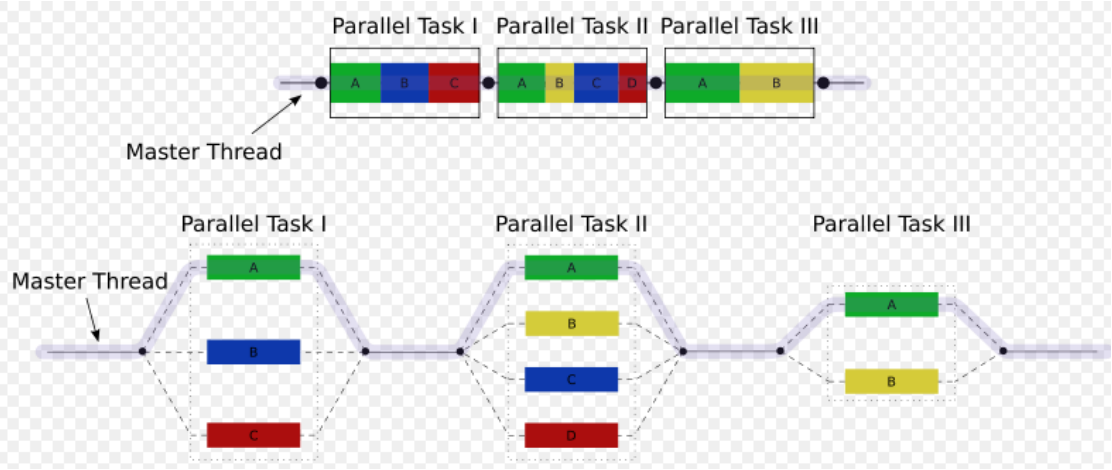
OpenMP

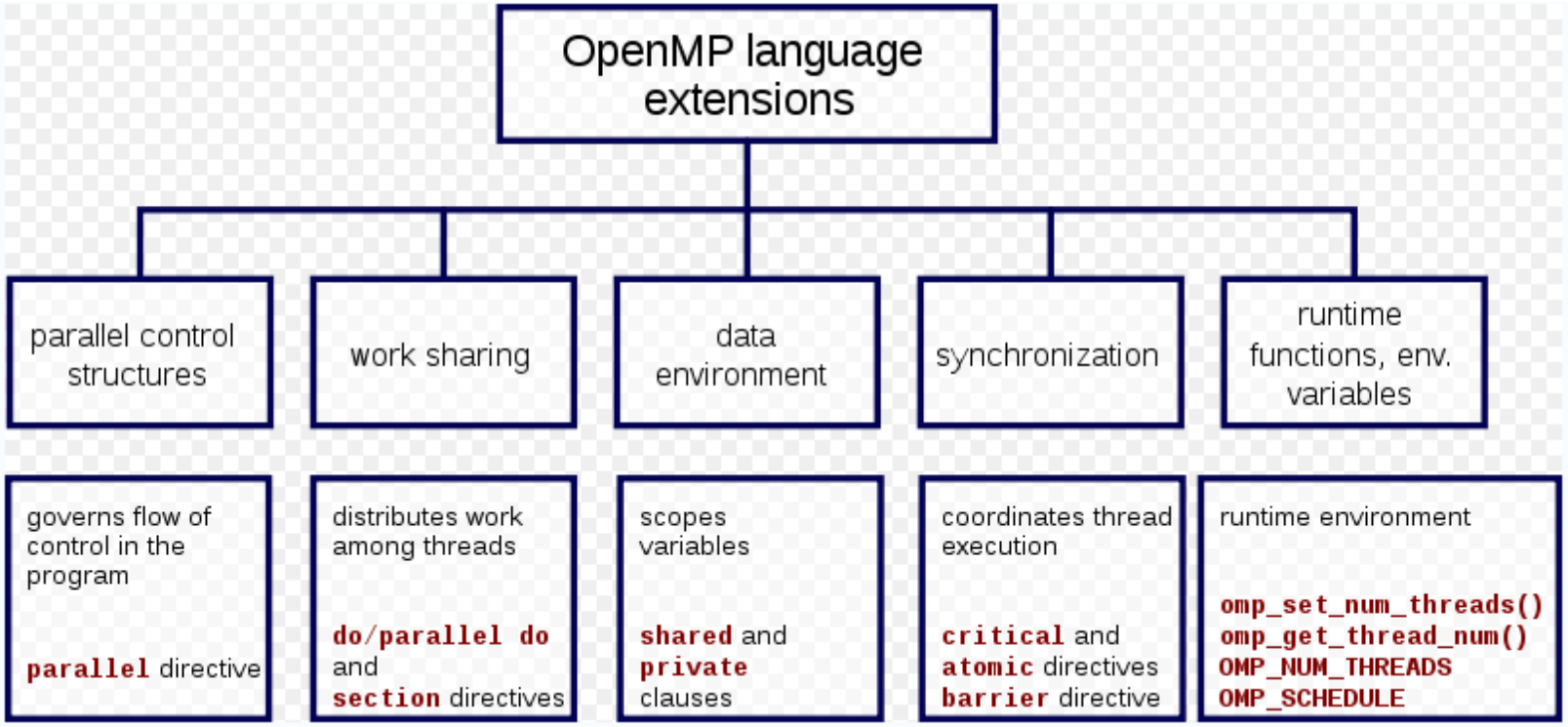
- OpenMP (Open Multi-Processing) is an API for multi-threaded shared memory programming
  - Provides compiler directives (~80%), library routines (~19%), and environment variables (~1%)
  - Needs compiler support
  - Header `omp.h`
  - Add `-fopenmp` to the gcc options on Linux
- Evolving standard (first specification in 1997, version 5.1 in 2020)

OpenMP

- Standardization and portability
  - Standard across a variety of shared memory architectures and platforms
  - Supports Fortran, C and C++
  - Scalable from embedded systems to the supercomputer
- Ease of use
  - *Simple* and *limited* set of directives → 3 or 4 are enough to implement significant parallelism
  - *Incremental* parallelization of a serial program
  - Coarse-grained and fine-grained parallelism

- OpenMP is based on the fork-join paradigm:
  - A *master* thread forks a specified number of *slave* threads
  - Tasks are divided among slaves
  - Slaves run concurrently as the runtime environment allocates threads to different processors




OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

OpenMP C/C++ Syntax

- Preprocessor directives called pragma (pragmatic information).
  - They are the main part of the OpenMP standard
  - Identify tasks and their synchronizations

```
#pragma omp <name> [list of clauses]
```

- Auxiliary C functions
  - Used to set and get relevant information such as number of available threads
  - Used to manage explicit locks

- Environment variables

Parallel control structures


```
#pragma omp parallel
{
        /* parallel section */
}
```

- OpenMP programs execute serially until they reach a `parallel` directive
  - The thread that was executing the code spawns a group of slave threads and becomes the master (thread ID 0)
  - The code in the structured block is replicated, each thread executes a copy
  - At the end of the block there is an implied barrier, only the master thread continues

Parallel control structures

- Optional clauses to the `parallel` directive:
  - Conditional parallelization with `if(condition)`
  - Number of spawned threads with `num_threads(int)`
  - Data scope clauses (see later)
- Under the hood, your compiler might replace the directive with a Pthreads implementation

```
#pragma omp parallel num_threads (8)
```

```
for (i = 0; i < 8; i++)
  pthread_create(...);
for (i = 0; i < 8; i++)
  pthread_join(...);
```

Parallel control structures

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the `if` clause
  - Value of the `num_threads` clause
  - Use of the `omp_set_num_threads()` library function
  - Setting of the `OMP_NUM_THREADS` environment variable
  - Implementation default, e.g., the number of CPUs on a node.

Work sharing

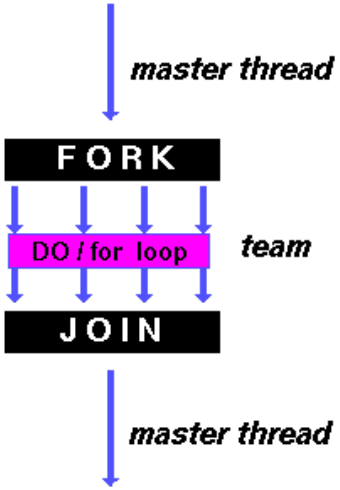
- Work-sharing constructs divide the execution of a code region among the members of the team that encounter it
  - A work-sharing construct must be *enclosed within* a `parallel` region for the directive to execute in parallel
  - Work-sharing constructs do not launch new threads
  - There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct

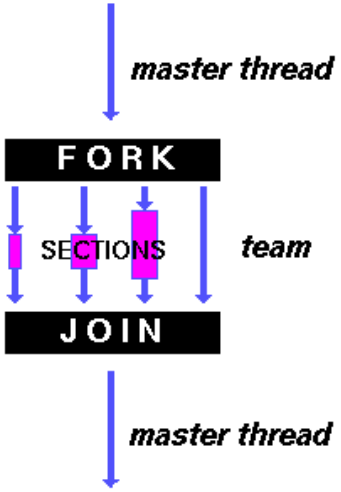
Work sharing

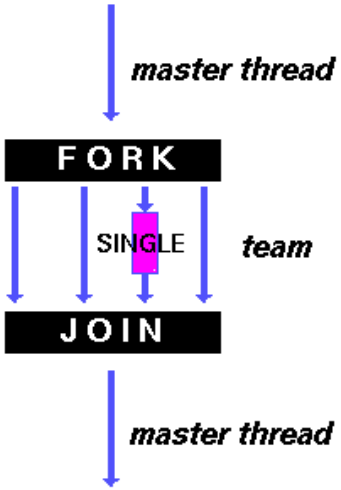
for: shares iterations of a loop across the team (data parallelism)

sections: breaks work into separate, discrete sections, each executed by a thread (functional parallelism)

single/master: serializes a section of code.







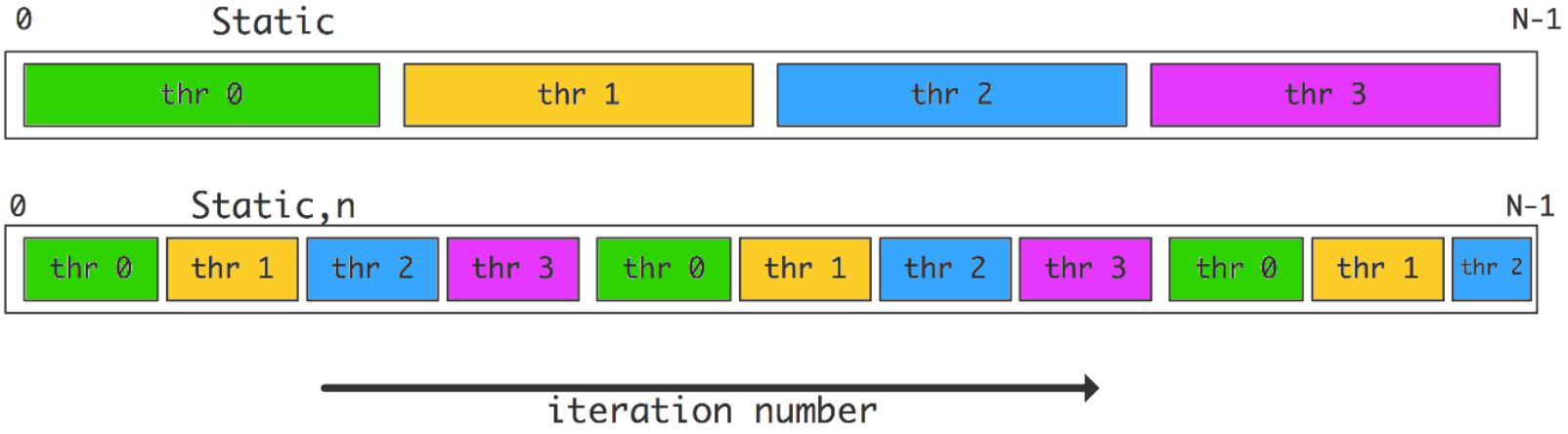
Work sharing


```
#pragma omp parallel
{
        #pragma omp for
        /* for loop */
}
```

- Parallelize execution of iterations
  - Iterations number cannot be internally modified
- Possible clauses include:
  - `schedule` describing how iterations of the loop are divided among the threads in the team
  - `nowait` to avoid synchronizing at the end of the parallel loop
  - Data-scope clauses (see later)

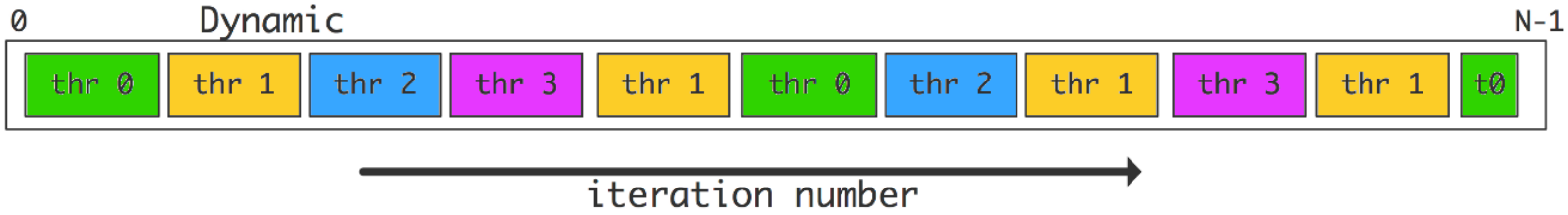
Work sharing

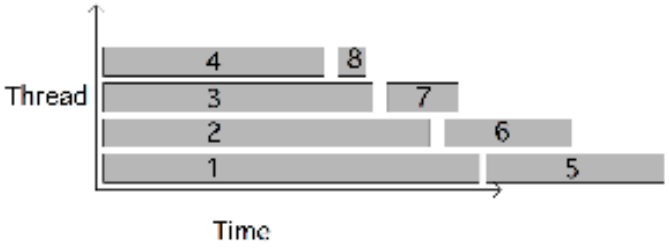
- Most typical `for` schedules
  - `static`: loop iterations are divided into blocks of size `chunk` and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads



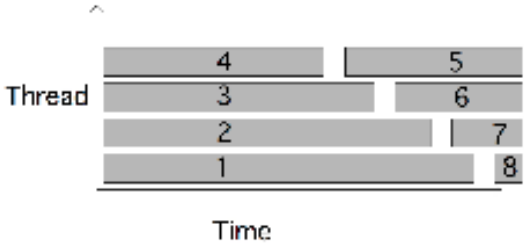
Work sharing

- Most typical `for` schedules
  - `dynamic`: loop iterations are divided into blocks of size `chunk` and distributed at runtime among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1





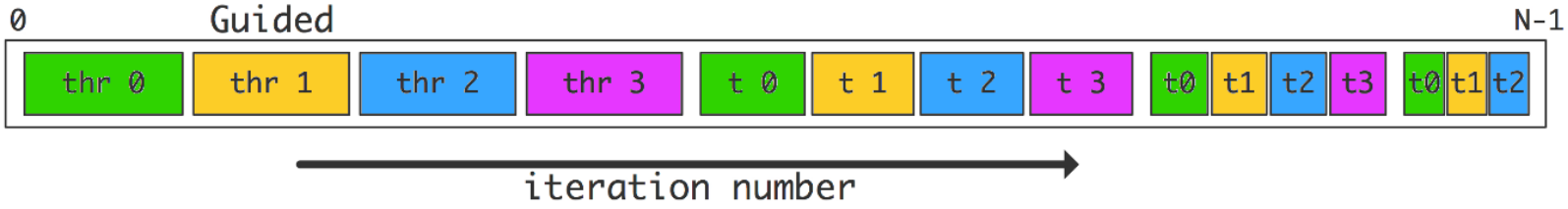
Static



Dynamic

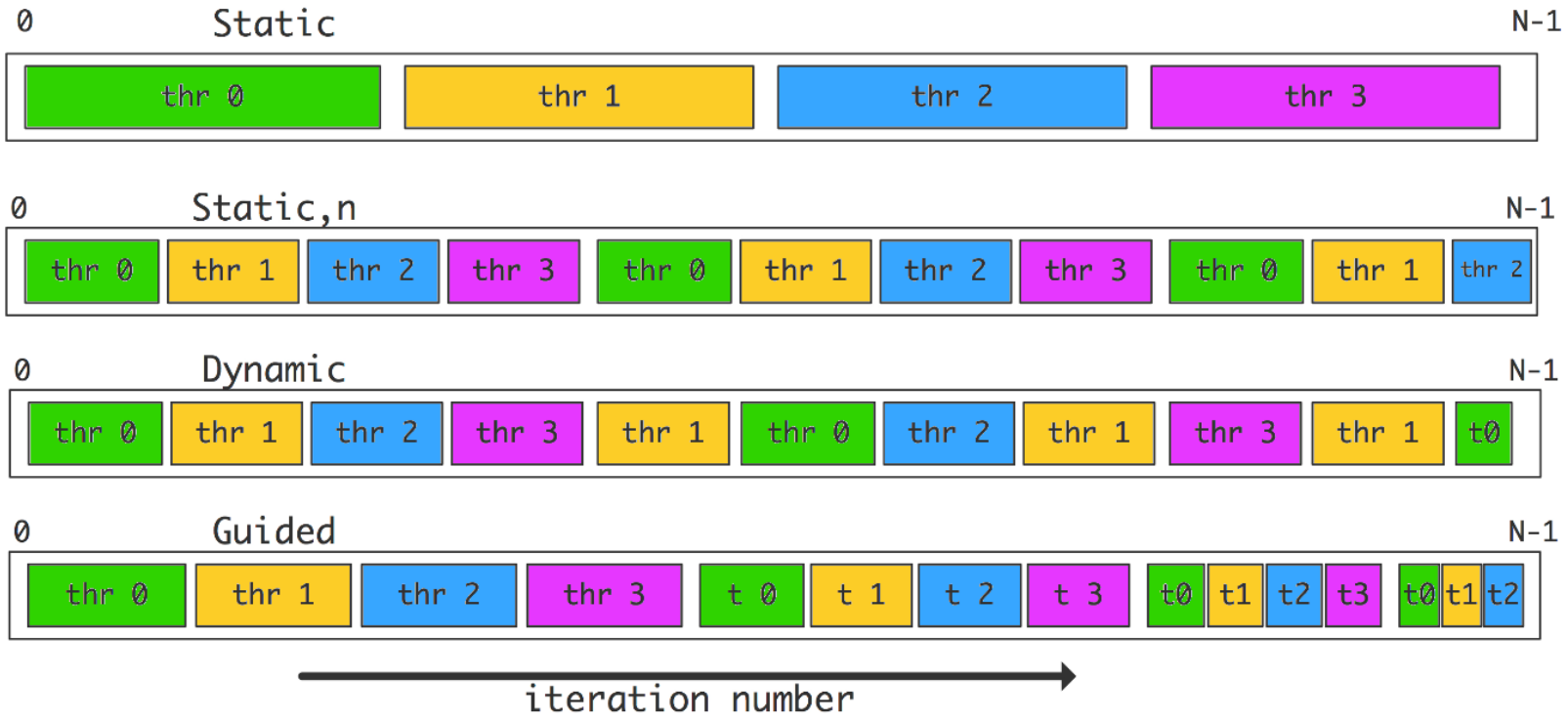
Work sharing

- Most typical `for` schedules
  - `runtime`: depends on the environment variable `OMP_SCHEDULE`
  - guided: static, gradually decreases the chunk size (`chunk` specifies the smallest one)



Work sharing

- Trade-off between low overhead (large chunks, static scheduling) and load balancing (small chunks, dynamic scheduling)



Work sharing


```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      /* code section 1 */
    }
    #pragma omp section
    {
      /* code section 2 */
    }
  }
}
```

- Specifies that the enclosed section(s) of code are to be executed in parallel
- Each `section` is executed once by a thread in the team

Work sharing


```
#pragma omp parallel
{
        #pragma omp single
        {
                /* code section */
        }
        #pragma omp master
        {
                /* code section */
        }
}
```

- `single` specifies that a section of a code is executed only by a single thread
- `master` specifies that a section of a code is executed only by the master

Work sharing


```
#pragma omp task
{
        /* code section */
}
#pragma omp taskwait
#pragma omp taskyield
```

- The `task` directive specifies a work unit which may be executed or deferred to another thread in the same team
- The `taskwait` directive introduces a barrier
- The `taskyield` directive interrupts execution of the current task
  - May be resumed by the same thread (tied clause) or by another (untied)

Synchronization


```
#pragma omp critical [name]
{
        /* code section */
}
```

- The `critical` directive specifies a region of code that must be executed by only one thread at a time
- The optional `name` enables multiple different `critical` regions:
  - names act as global identifiers: different `critical` regions with the same name are treated as the same region
  - all `critical` sections which are unnamed are treated as the same section

Synchronization


```
#pragma omp barrier
```

- The `barrier` directive synchronizes all threads in the team
- When a `barrier` directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier
- Not very used because of implicit synchronization of other constructs

Synchronization


```
#pragma omp atomic
        /* statement */
```

- The `atomic` directive ensures that a specific storage location is accessed atomically
- Multiple reads and writes are not allowed
- Only valid for the following statement, not for a structured block

Data environment

- OpenMP is based upon the shared memory programming model so most variables are **shared by default**
- The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include:
  - `private`
  - `shared`
  - `default`
  - `reduction`

Data environment

- Data Scope Attribute Clauses are used in conjunction with several directives to
  - define how and which data variables in the serial section of the program are transferred to the parallel sections
  - define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads

Data environment


```
#pragma omp <name> private (list)
```

- The `private` clause declares variables in its list to be private to each thread
- `private` variables behave as follows:
  - a new object of the same type is declared once for each thread in the team
  - all references to the original object are replaced with references to the new object
  - assume that each variable is uninitialized for each thread – or use `firstprivate`

Data environment


```
#pragma omp <name> shared (list)
```

- The `shared` clause declares variables in its list to be shared among all threads in the team
- A `shared` variable exists in only one memory location and all threads can read or write to that address
- It is the programmer's responsibility to ensure that multiple threads properly access `shared` variables

Data environment

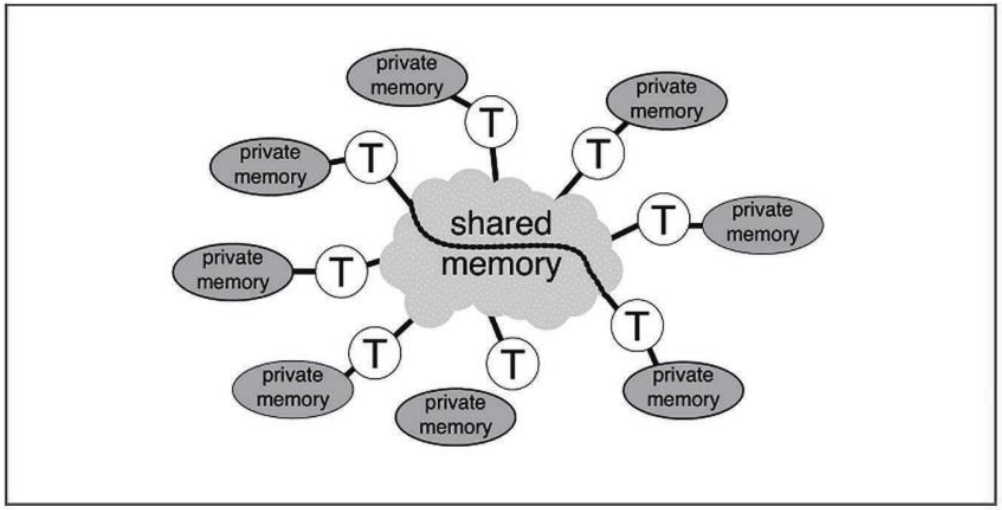

```
#pragma omp <name> default (shared | none)
```

- The scope of *all* variables is set to `shared` or `none`
  - `default(shared)` is already the default, so it can be omitted
- Specific variables can be exempted from the default using specific clauses (`private`, `shared`, etc.)
- Using `none` as a default requires that the programmer explicitly scope all variables
- In some implementations `default (private)` is also an option

Data Environment & Work sharing

- A reduction variable in a loop aggregates (e.g., accumulates) a value that:
  - depends on each iteration of the loop
  - does not depend on the iteration order
- The `reduction (operator: list)` clause helps to perform a reduction
  - A private copy of each variable in the list is updated by each thread
  - At the end the reduction operation is applied to all private copies and the end result is written into a global variable
  - Available operators: + * - & | ^ && || min max

The OpenMP memory model



- Threads have private memory and they can access a shared memory (single address space)
- Variables are scoped through appropriate clauses
- What happens when a thread modifies a shared variable? When do the other threads see the modified value? → Memory consistency model

The OpenMP memory model


```
<Shared variables x and y are initialized to zero>

     Thread 0                    Thread 1

     x = 1;                      y = 1;
     if ( y == 0 )               if ( x == 0 )
       function_call();            function_call();
```

- When does Thread 0 see the modification to `y`? When does Thread 1 see the modification to `x`?
- The compiler does not know how threads will interact. What happens if it moves the assignment to `x` after the if construct?
- In what order are variables stored in the main memory?

The OpenMP memory model


```
<Shared variables x and y are initialized to zero>

     Thread 0                    Thread 1

     x = 1;                      y = 1;
     if ( y == 0 )               if ( x == 0 )
       function_call();            function_call();
```

- Sequential memory consistency: each thread performs loads and stores in the original sequential order, and stores are atomic
- Only one thread executes `function_call`
- Excludes optimizations that move instructions, limits performance

The OpenMP memory model

- OpenMP employs a **relaxed** memory consistency model, i.e., all threads have the same view of memory *at specific points in the code* (consistency points)
- In between such points each thread has its own *temporary* view of memory which may be different from the temporary view of other threads
- Read-only data → no consistency issues
- Shared data that needs to be modified → possible data races
- The user needs to know *when a shared variable may be read reliably*

The OpenMP memory model


```
sum = 0;
#pragma omp parallel shared(sum)
{

  int my_contribution; // Contains the per-thread partial sum
    ....
  my_contribution = ....  // Thread computes a value

  // Without the critical region this code has a data race

  #pragma omp critical
  {
      sum += my_contribution;
  } // End of critical region
    ....
} // End of parallel region
```

With a relaxed consistency model, how do I know the next thread is going to read the most up-to-date version of sum?

The OpenMP memory model


```
#pragma omp flush [flush-set]
```

- The `flush` directive enforces global consistency of shared variables
- Between a `flush` and the next update of shared variables all threads are guaranteed to have the same global view of all shared memory
- Without `flush-set` all shared variables are affected; if possible, *don't use* a `flush-set`
  - Compilers are allowed to move flush constructs if they have a disjoint set
  - Difficult to use correctly, invalid behavior

The OpenMP memory model


```
#pragma omp flush [flush-set]
```

- All threads must execute the same `flush` to guarantee consistency
- To simplify development, a `flush` construct is implied:
  - During a `barrier` region
  - At entry and exit of `parallel` and `critical`
  - At exit from worksharing constructs, unless `nowait` is specified
- A `flush` construct is not implied
  - At entry of worksharing constructs
  - At entry and exit of `master`

The OpenMP memory model


```
#pragma omp flush

while ( execution_state[i] != READ_FINISHED ) {

   <wait for a short while>

   #pragma omp flush

} // End of while-loop
```

Ensures we read `execution_state[i]` correctly

Avoids hanging forever

Assuming `master` controls the value of `execution_state[i]`, it will also need a `flush` after changing it

Runtime functions

- The OpenMP standard defines an API for library calls that perform a variety of functions to control execution of the program

- **`int omp_get_num_threads()`**
  - Returns the number of threads that are currently in the team executing the parallel region from which it is called
- **`int omp_get_thread_num()`**
  - Returns an identifier for the thread making this call. This number will be between 0 and `omp_get_num_threads` - 1. The master thread of the team is thread 0

Runtime functions

- `void omp_set_num_threads(int num_threads)`
  - Sets the number of threads that will be used in the next parallel region
- `double omp_get_wtime()`
  - Provides a wall clock timing routine
- `double omp_get_wtick()`
  - Returns the number of seconds between two clock ticks

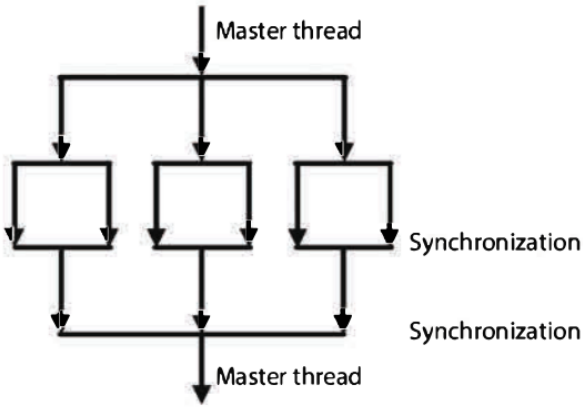
Environment variables

- Environment variables are set to control execution of all programs, for example with the `export` command in bash
- `OMP_SCHEDULE`
- `OMP_NUM_THREADS`
- `OMP_NESTED`
- `OMP_STACKSIZE`
- `OMP_WAIT_POLICY`
- `OMP_PROC_BIND`

Nested parallelism

- OpenMP `parallel` constructs can be nested to achieve different levels of parallelism
- Each thread that encounters a new `parallel` region spawns new threads

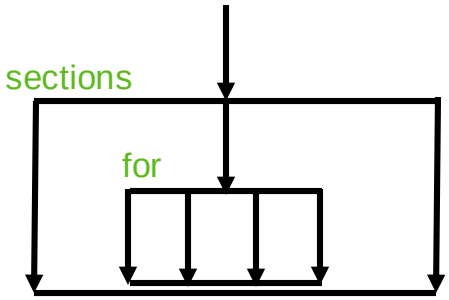
```
#pragma omp parallel num_threads(3)
{
   <code in here is executed by 3 threads>
   #pragma omp parallel num_threads(2)
   {
       <code executed by 3x2 = 6 threads>
   } // End of second level parallel region
} // End of first level parallel region
```



Nested parallelism

- Parallel worksharing constructs can also be nested

```
pragma omp parallel sections
{
  #pragma omp section
  { printf("I am section 1\n"); }
  #pragma omp section
  {
     printf("I am section 2\n");
     #pragma omp parallel for shared(n) num_threads(4)
     for (int i=0; i<n; i++)
     {
         printf("Section 2:\tIteration = %d Thread ID = %d\n",
                 i, omp_get_thread_num());
     } // End of parallel for loop
  }
  #pragma omp section
     { printf("I am section 3\n"); }

} // End of parallel sections
```



Nested parallelism

- Not all OpenMP implementations include nested parallelism
- Useful variable: `OMP_DISPLAY_ENV`, if set to `true`, prints all settings at program start
  - If set to `verbose`, adds information about vendor-specific extensions

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP='201307'
  OMP_CANCELLATION='FALSE'
  OMP_DISPLAY_ENV='true'
  OMP_DYNAMIC='TRUE'
  OMP_MAX_ACTIVE_LEVELS='4'
  OMP_NESTED='FALSE'
  OMP_NUM_THREADS='16'
  OMP_PLACES='N/A'
  OMP_PROC_BIND='FALSE'
  OMP_SCHEDULE='static'
  OMP_STACKSIZE='8388608B'
  OMP_THREAD_LIMIT='1024'
  OMP_WAIT_POLICY='PASSIVE'
OPENMP DISPLAY ENVIRONMENT END
```

Nested parallelism

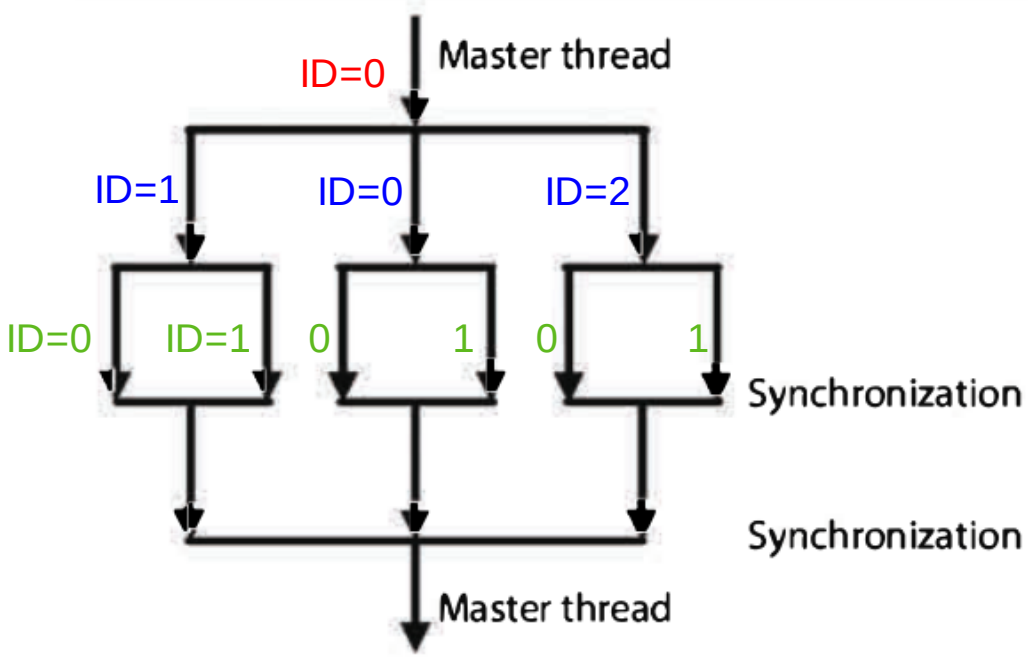
- Enable/disable nested parallelism
  - Through the runtime function `omp_set_nested()`
  - Through the environment variable `OMP_NESTED`
  - Defaults are implementation-dependent
- Set a default number of threads at different levels of nested parallelism with `OMP_NUM_THREADS=<list,of,integers>`
  - If the nesting level is deeper than the number of entries in the list, the last value is used for all subsequent nested parallel regions.

Nested parallelism

- `OMP_MAX_ACTIVE_LEVELS` defines the upper limit on the number of active parallel regions that may be nested
- `OMP_THREAD_LIMIT` avoids that recursive applications create too many threads

Nested parallelism

- Each thread starts a new team, so thread IDs start from 0 again!
- `omp_get_thread_num()` is not enough to identify a thread



Nested parallelism

- Runtime functions for nested parallelism
  - `omp_get_thread_limit`
  - `omp_get_max_active_levels`
  - `omp_set_max_active_levels(int max_levels)`
  - `omp_get_level`
  - `omp_get_active_level`
    - *active* regions only (i.e., with more than one thread)
  - `omp_get_ancestor_thread_num`
  - `omp_get_team_size`

Exercises

https://drive.google.com/file/d/100J_bPBEp5S_5dFJK6qvoh-hk6MPNvfp/view?usp=sharing

Hello World - OpenMP


```
int main ()
{
   #pragma omp parallel
   {
      int id = omp_get_thread_num();
      printf("Hello World from thread = %d\n", id);
   }
}
```

- What do you expect will be printed?
- How many threads are created?
- What happens if you ask for a number of threads greater than the number of cores?

Remember how the PThreads version of the same program looked like?

Incremental parallelization

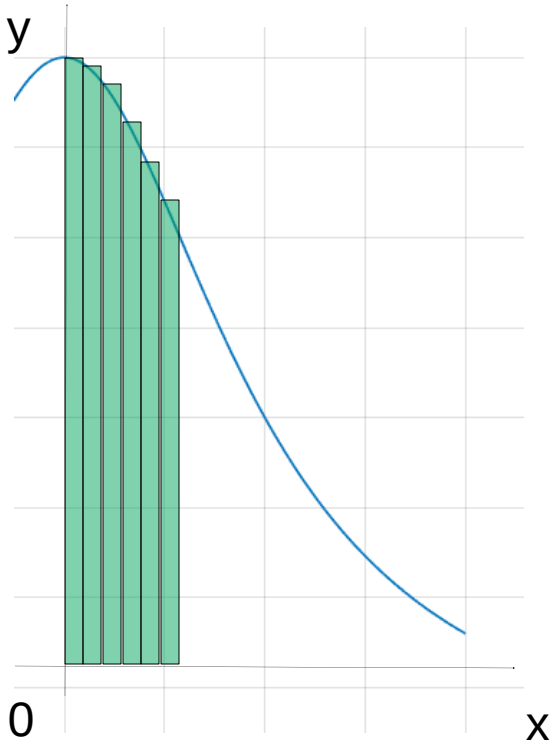

```
void print_message(int threadIndex) {
  printf("Thread number %d\n", threadIndex);
}

int main() {
  #pragma omp parallel num_threads(4)
  {
    #pragma omp for schedule(static, 4)
    for (unsigned int ii = 0; ii < 10; ii++) {
      print_message(ii);
    }
  }
  return 0;
}
```

- This program is valid with and without OpenMP
- Try the following:
  - Add and remove -fopenmp
  - Inspect the generated compiler IR
  - Run the program

Calculation of pi - integral

- Approximate the integral as the sum of the areas of small rectangles
- Each thread calculates the height of a set of rectangles (map)
- The sum of all heights is multiplied by the step size to get the area

$$\int_0^1 \frac{4}{1+x^2} = \pi$$



Calculation of pi - integral

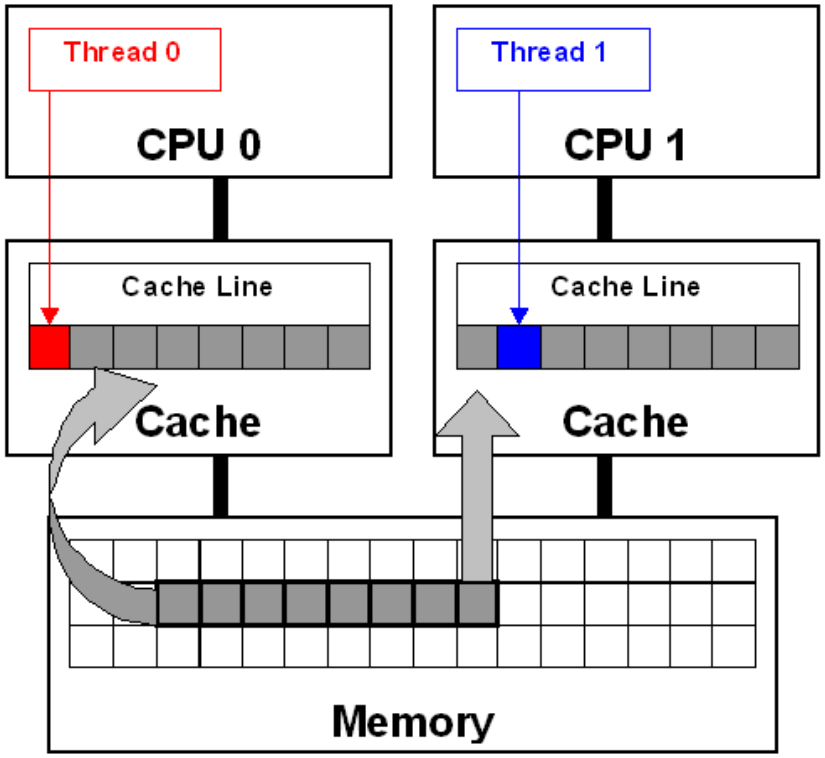

```
static long num_steps = 100000000;
double step;
int main ()
{
  double pi, full_sum = 0.0;
  double start_time, run_time;
  double sum[4];
  step = 1.0/(double) num_steps;
  omp_set_num_threads(4);
  full_sum=0.0;

  #pragma omp parallel
  {
    int i;
    int id = omp_get_thread_num();
    int numthreads = omp_get_num_threads();
    double x;
    sum[id] = 0.0;

    for (i=id; i<num_steps; i+=numthreads){
      x = (i+0.5)*step;
      sum[id] = sum[id] + 4.0/(1.0+x*x);
    }
  }
```

```
      for(i=0; i<4; i++)
        full_sum += sum[i];

      pi = step * full_sum;
      printf("\n pi is %f \n",pi);
}
```

- How is work distributed?
- Is the result deterministic?
- How is access to the shared variable solved?
- What happens if we increase the number of threads?

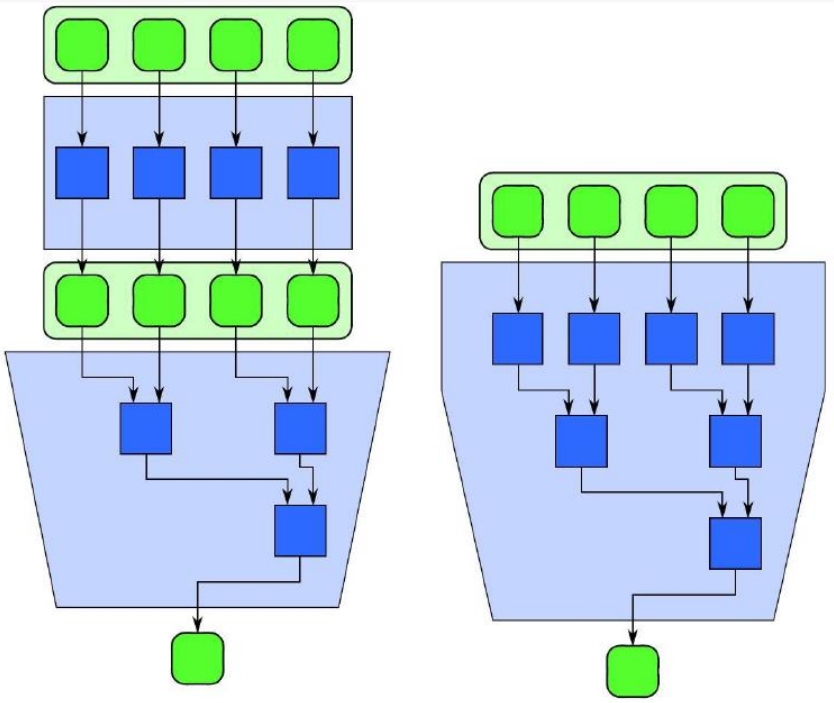
Calculation of pi - integral

- Performance does not scale as we would expect → false sharing issue



Calculation of pi - integral

- Solution 1: add synchronization
- Solution 2: use a work-sharing construct
  - (map + reduction)

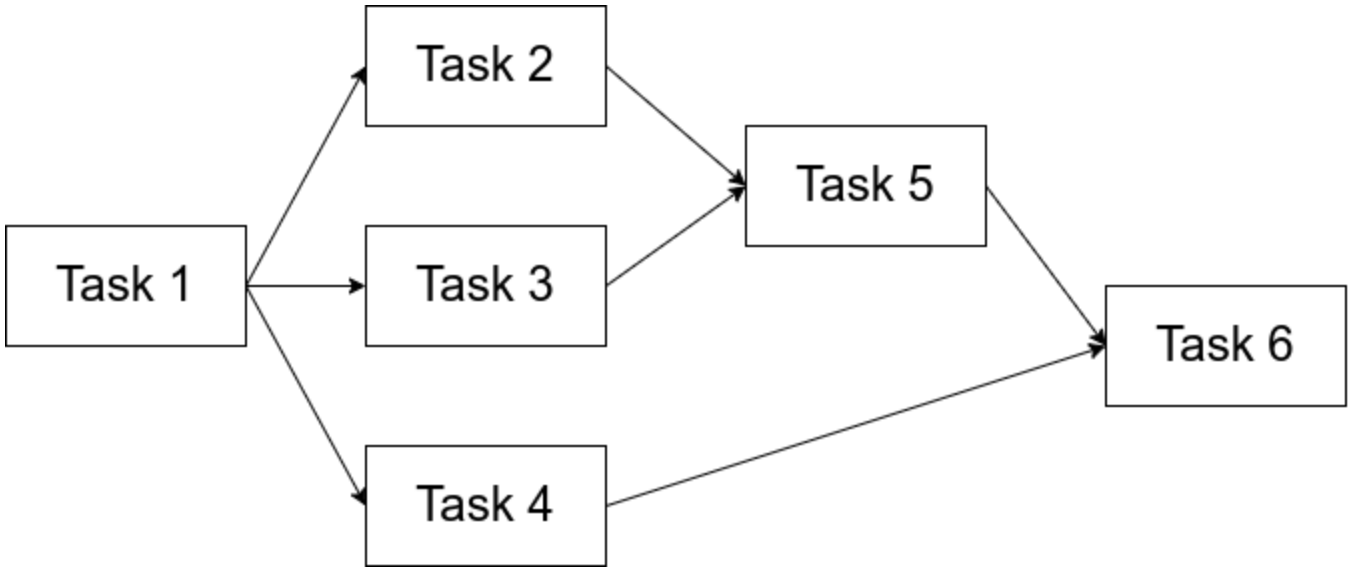


Calculation of pi - integral

- Version with `parallel for` and `reduction`

```
static long num_steps = 100000000;
double step;
int main ()
{
   double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;
   sum = 0.0;
   omp_set_num_threads(4);

   #pragma omp parallel for private(x) reduction(+:sum)
   for (i=1;i<= num_steps; i++)
   {
      x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }

   pi = step * sum;
}
```

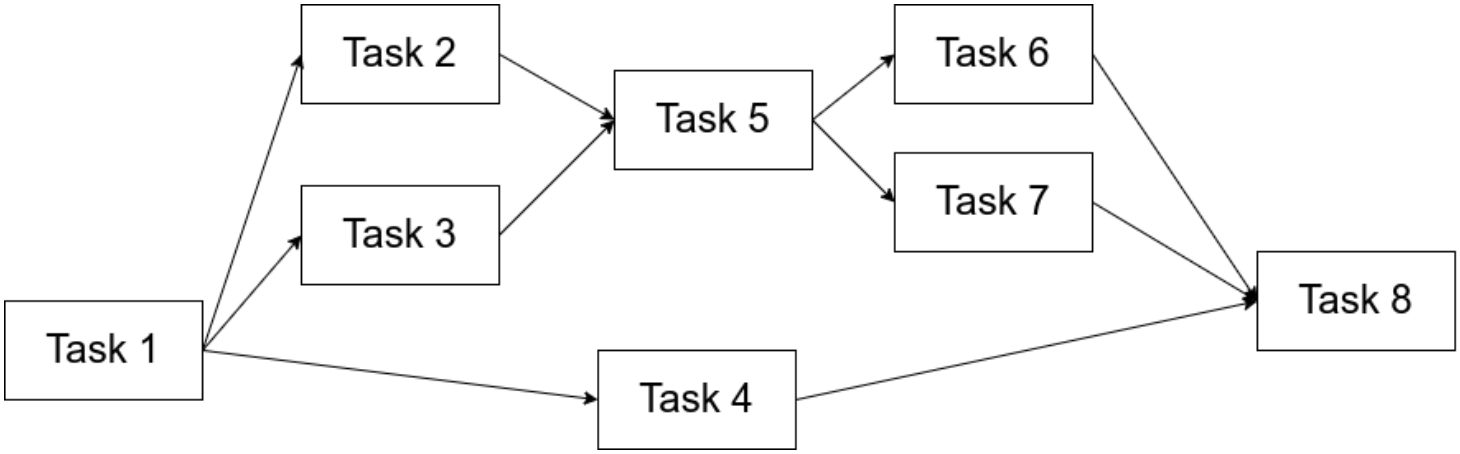
Task Graph Exercises/1



W(T1) = 100
W(T2) = 90
W(T3) = 50
W(T4) = 250
W(T5) = 70
W(T6) = 300

- Calculate work and parallelism.
- Write an OpenMP implementation reflecting the structure of the task graph.
- How many threads are active during the execution of Task 5?
- Is there a better parallel implementation (considering both performance and resource usage)?

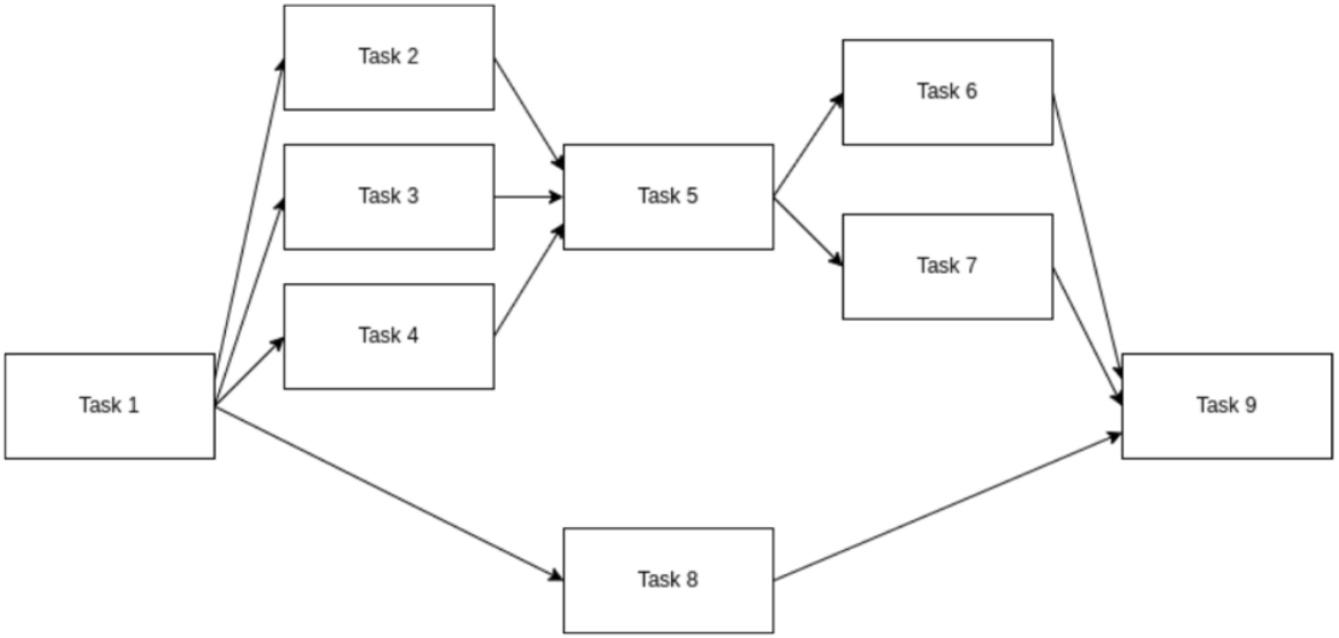
Task Graph Exercises/2



W(T1) = 100
W(T2) = 100
W(T3) = 75
W(T4) = 50
W(T5) = 70
W(T6) = 300
W(T7) = 250
W(T8) = 100

- Calculate work and span.
- Write an OpenMP implementation reflecting the structure of the task graph.
- Is this implementation faster than a sequential one? How much?

Task Graph Exercises/3



W(Task1) = 50,
W(Task2) = 200,
W(Task3) = 250,
W(Task4) = 200,
W(Task5) = 50,
W(Task6) = 100,
W(Task7) = 200,
W(Task8) = 450,
W(Task9) = 50

- Is it better to implement the program with 3 or 4 threads?
- Write a corresponding OpenMP implementation.
- If you could optimize one task by bringing its work to 0 (maintaining the graph structure), which one would yield the best improvement in terms of parallel execution time?

Credits & References

- I.Foster, "Designing and Building Parallel Programs", Addison-Wesley, 1995
- Video series from Tim Mattson (Intel) Introduction to OpenMP
- Ruud van der Pas, Eric Stotzer, and Christian Terboven, "Using OpenMP-The Next Step. Affinity, Accelerators, Tasking, and SIMD", MIT Press 2017
- Victor Eijkhout, "The art of HPC", https://theartofhpc.com

Credits & References

- *Introduction to Parallel Computing, POSIX Threads Programming*, and *OpenMP Tutorial* by Blaise Barney, Lawrence Livermore National Laboratory https://hpc.llnl.gov/training/tutorials
- UC Berkeley CS267: *Applications of Parallel Computers* https://sites.google.com/lbl.gov/cs267-spr2020
- "Introduction to Parallel Computing", Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar https://sites.google.com/lbl.gov/cs267-spr2020