OPENMP BASICS

OpenMP Basics

- **Outline**
  - Overview
  - Execution Model
  - Threads and Data Access
  - Application Program Interface
  - Thread Synchronization
  - Memory Consistency

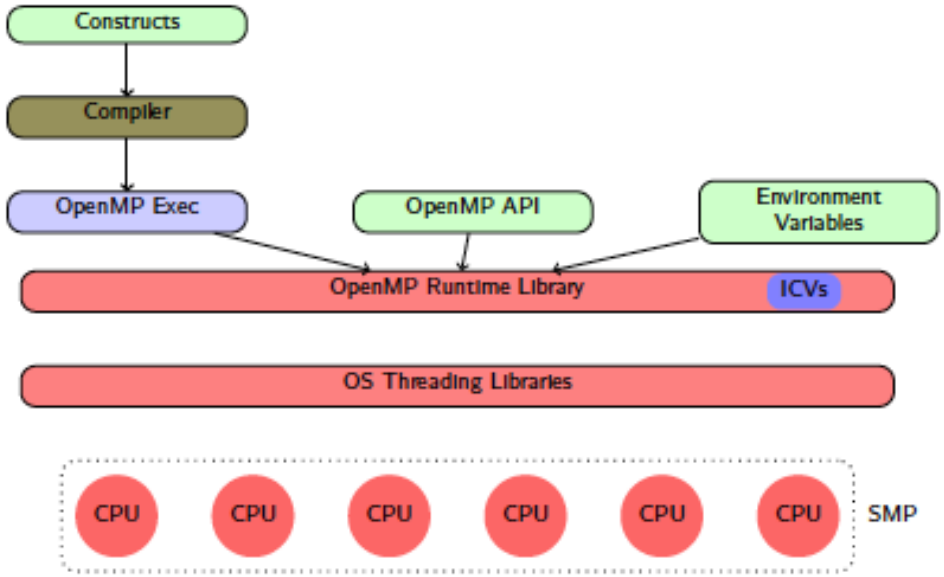
OpenMP Basics

What is OpenMP?

- It's an API extension to the C, C++ and Fortran languages to write parallel programs for shared memory machines
- Current version is 5.0 (November 2018)
- Supported by most compiler vendors
  - ✓ Intel, IBM, HP, GCC,...
- Maintained by the Architecture Review Board (ARB), a consortium of industry and academia
- This mini-tutorial just covers part of the specification, for a complete reference please consult the documentation online
  - ✓ http://www.openmp.org

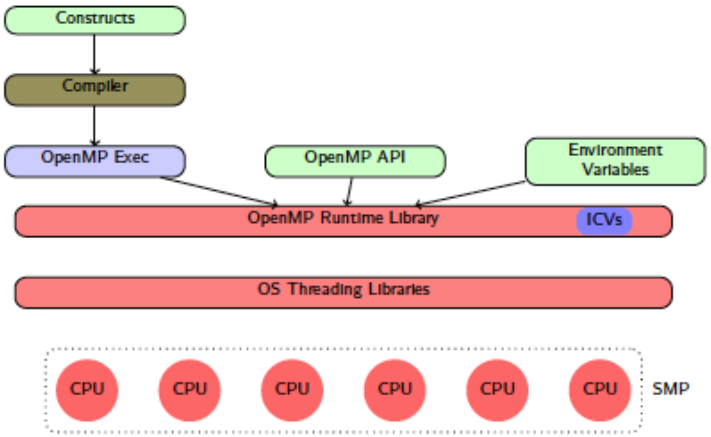
OpenMP Basics

- **OpenMP Components:**



OpenMP Basics

- **OpenMP Components:**
  - Constructs
    - ✓ These form the major elements of OpenMP programming
      - – Create threads
      - – Share the work amongst threads
      - – Synchronize threads and memory



  - Library routines
    - ✓ To control and query the parallel execution environment (internal control variables - ICVs)
  - Environment variables
    - ✓ The execution environment can also be set before the program execution is started

OpenMP Directives

- **OpenMP directives syntax**
  - In C/C++, through a compiler directive:
    - ✓ `#pragma omp [optional clauses]`
  - OpenMP syntax is ignored if specific flag is not set for the compiler

- **Most directives apply to:**
  - A block of one or more statements
    - ✓ One entry point, one exit point
      - – No branching in or out allowed

OpenMP Headers and Macros

- **C/C++ only**
  - `omp.h` contains the API prototypes and data types definitions
  - The `_OPENMP` macro is defined by OpenMP enabled compiler
    - ✓ Allows conditional compilation of OpenMP

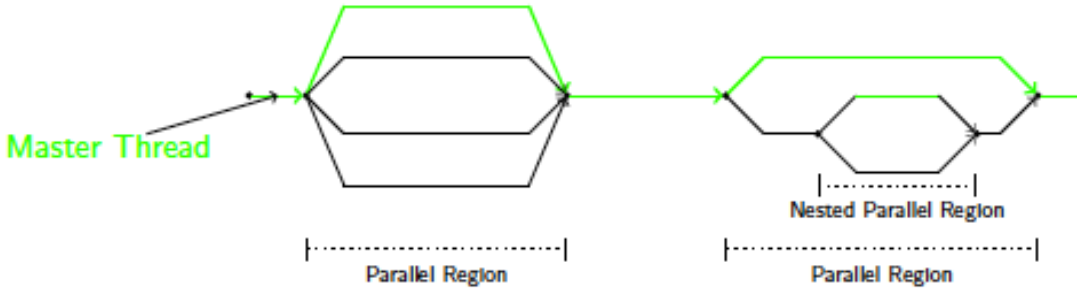
```
// Conditional code for OpenMP

#ifdef _OPENMP
       //  OpenMP code
#else
       // Serial, original code without OpenMP
#endif
```

Performance Modeling

- **Outline**
  - Overview
  - Execution Model
  - Threads and Data Access
  - Application Program Interface
  - Thread Synchronization
  - Memory Consistency

OpenMP Execution Model

- **Fork-join model**
  - OpenMP uses a fork-join model
    - ✓ The master thread spawns a team of threads that joins at the end of the parallel region
    - ✓ Threads in the same team can collaborate to do work



OpenMP Memory Model

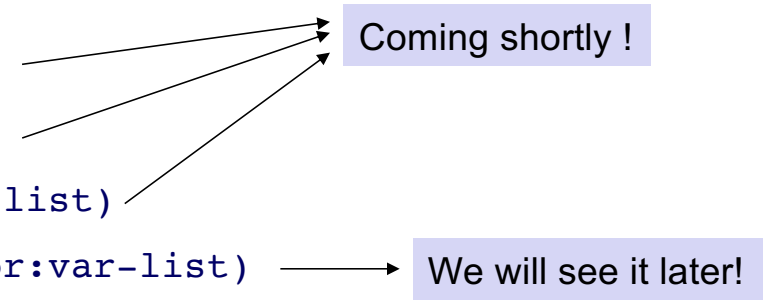
- **OpenMP defines a relaxed memory model**
  - Threads can see different values for the same variable
  - Memory consistency is only guaranteed at specific points
  - Luckily, the default points are usually enough
    - ✓ If not ... there is a mechanism to guarantee it! (described at the end of Part I)
  - Variables can be shared or private to each thread

Performance Modeling

- **Outline**
  - Overview
  - Execution Model
  - Threads and Data Access
  - Application Program Interface
  - Thread Synchronization
  - Memory Consistency

Parallelism Definition

- **The `parallel` construct**
  - Directive
    - ✓ `#pragma omp parallel [ c l a u s e s ]`
      - `structured block`
  - Where some of the clauses are:
    - ✓ `num threads(expression)`
    - ✓ `if(expression)`
    - ✓ `shared(var-list)`
    - ✓ `private(var-list)`
    - ✓ `firstprivate(var-list)`
    - ✓ `reduction(operator:var-list)`



Parallelism Definition

- **The `parallel` construct**
  - Specifying the number of threads
    - ✓ The nthreads-var ICV is used to determine the number of threads to be used for encountered parallel regions
    - ✓ It is a list of positive integer values, its first element specifying the number of processors for the next nesting level
    - ✓ When a parallel construct is encountered, and the generating task's `nthreads-var` list contains multiple elements, the generated task(s) inherit the value of `nthreads-var` as the list obtained by deletion of the first element
    - ✓ If the generating task's `nthreads-var` list contains a single element, the generated task(s) inherit that list as the value of `nthreads-var`

Parallelism Definition

- **The `parallel` construct**
  - Specifying the number of threads
    - ✓ The `nthreads-var` list can be defined from the execution command line by setting the `OMP_NUM_THREADS` environment variable
    - ✓ Example: `setenv OMP_NUM_THREADS 4,3,2`
    - ✓ After that, the `threads-var` list can be modified through:
      - the `omp set_num_threads` API, which sets the value of the first element of the current list
      - the `num_threads` clause, which causes the implementation to ignore the ICV and use the value of the clause for that region

Parallelism Definition

- **The `if` clause**
  - Avoiding parallel regions
    - ✓ Sometimes we only want to run in parallel under certain conditions (e.g.: enough input data, not running already in parallel, ...)
  - The if clause allows to specify an expression. When evaluates to false the parallel construct will only use 1 thread
  - Note that still creates a new team and data environment

Parallelism Definition

- **The `parallel` construct**
  - How many threads are used in each parallel region below?

```
void main ( ) {

#pragma omp parallel
. . .
omp_set_num_threads ( 2 );

#pragma omp parallel
. . .
#pragma omp parallel num_threads (random()%4+1) if ( 0 )
. . .

}
```

Data Sharing Attributes

- **The `shared` clause**
  - When a variable is marked as shared, the variable inside the construct is the same as the one outside the construct
    - ✓ In a parallel construct this means all threads see the same variable
      - – But not necessarily the same value!
  - Usually need some kind of synchronization to update them correctly
    - ✓ OpenMP has consistency points at synchronizations
  - By default, variables are implicitly shared

Data Sharing Attributes

- **The `private` clause**
- **When a variable is marked as `private`, the variable inside the construct is a *new* variable of the same type with an *undefined* value**
  - In a parallel construct this means all threads have a different variable
  - Can be accessed without any kind of synchronization

Data Sharing Attributes

- **The `firstprivate` clause**
  - When a variable is marked as `firstprivate`, the variable inside the construct is a *new* variable of the same type but initialized to the original variable value
    - ✓ In a parallel construct this means all threads have a different variable with the same initial value
    - ✓ Can be accessed without any kind of synchronization

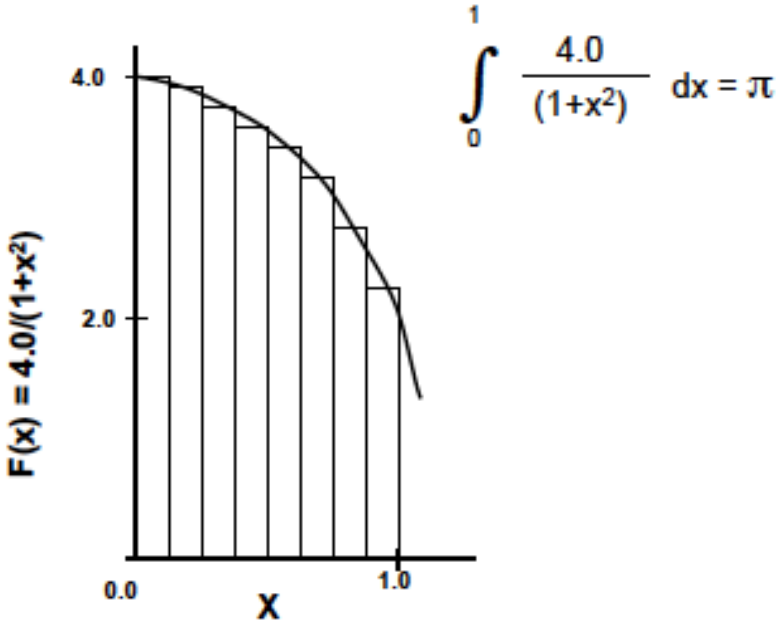
Data Sharing Attributes

- **Exercise**
  - What does appear on the screen if `XXXXXX` is `shared(x)`, `private(x)` or `firstprivate(x)`?

```
int x=1;
#pragma omp parallel XXXXXX num_threads(2)
{
  x++;
  printf("%d\n", x ) ;
}
printf("%d\n", x ) ;
```

Example: Computation of Pi

- **Implementation of PI computation with the following numerical recipe:**

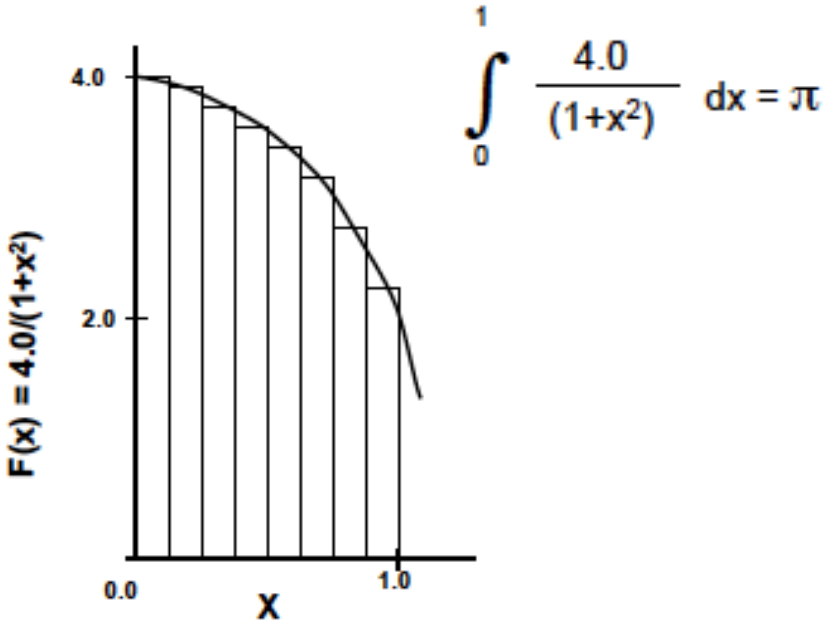


$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num steps = 100000;
double step;

void main ( )
{
  int i;
  double x , pi , sum = 0.0;
  step = 1.0 / (double) num_steps ;
  for (i=1; i<=num_steps; i++) {
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x );
  }
  pi = step*sum;
}
```

Example: Computation of Pi

- **Implementation of PI computation with the following numerical recipe:**



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num steps = 100000;
double step;
#include <omp.h>
#define NUM_THREADS(2)

void main ( )
{
  int i
  double x , pi , sum = 0.0;
  step = 1.0 / (double) num_steps ;

  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(x, i)
  {
    for (i=1; i<=num_steps; i++) {
      x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x );
    }
  }
  pi = step*sum;
}
```

Performance Modeling

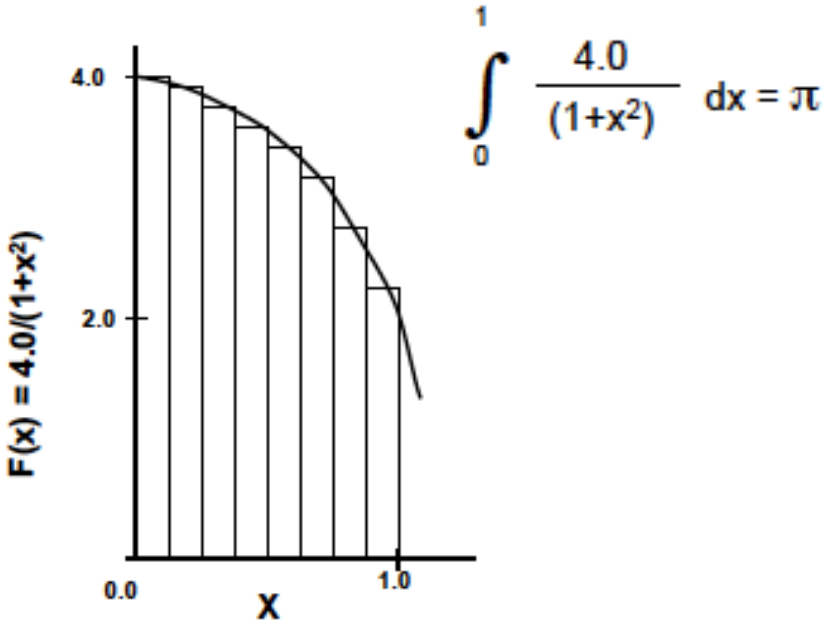
- **Outline**
  - Overview
  - Execution Model
  - Threads and Data Access
  - Application Program Interface
  - Thread Synchronization
  - Memory Consistency

Some API routines

- `int omp_get_num_threads()`
  - Returns the number of threads in the current team. Returns 1 if outside a parallel region
- `int omp get_thread_num()`
  - Returns the `id` of the thread in the current team. The `id` between 0 and `omp_get_num_threads()-1`
- `void omp_set_num_threads()`
  - Sets the number of threads to be used in parallel regions at the next nesting level
- `int omp_get_max_threads()`
  - Returns the number of threads that could be used in parallel regions at the next nesting level
- `double omp_get_wtime()`
  - Returns the number of seconds since an arbitrary point in the past

Example: Computation of Pi

- **Distribute iterations according to openmp `id`**

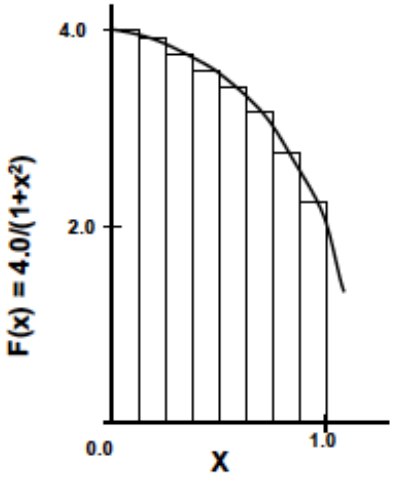


$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num steps = 100000;
double step;
#include <omp.h>
#define NUM_THREADS(2)

void main ( )
{
  int i
  double x , pi , sum = 0.0;
  step = 1.0 / (double) num_steps ;
  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(x, i)
  {
    id = omp_get_thread_num( ) ;
    for (i=id+1; i<=num_steps; i=i+NUM THREADS) {
      x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x );
     }
  }
  pi = step*sum;
}
```

Example: Computation of Pi

- **Measuring execution time**



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num steps = 100000;
double step;
#include <omp.h>
#define NUM_THREADS(2)

void main ( )
{
  int i
  double x , pi , sum = 0.0;
  double TimeStart, TimeStop;

  TimeStart = omp_get_wtime();

  step = 1.0 / (double) num_steps ;

  omp_set_num_threads(NUM_THREADS);

#pragma omp parallel private(x, i)
  {
    id = omp_get_thread_num( ) ;
    for (i=id+1; i<=num_steps; i=i+NUM THREADS) {
      x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x );
     }
  }
  pi = step*sum;
  TimeEnd = omp_get_wtime();
  printf("Wall clock time = %.20f\n", TimeEnd-TimeStart);
}
```

Performance Modeling

- **Outline**
  - Overview
  - Execution Model
  - Threads and Data Access
  - Application Program Interface
  - Thread Synchronization
  - Memory Consistency

Why synchronization ?

- **OpenMP follows a shared memory model**
  - Threads communicate by sharing variables
  - Unintended sharing of data causes race conditions (i.e. the execution outcome may change as the threads are scheduled differently)
  - Threads need to synchronize to impose some ordering in their sequence of actions
- **Some OpenMP synchronization mechanisms:**
  - `barrier`
  - `critical`
  - `atomic`
  - Use of locks through API

Thread Barrier

- **`#pragma omp barrier`**
  - Threads cannot proceed past a barrier point until all threads reach the barrier AND all previously generated work is completed
  - Some constructs have an implicit barrier at the end
    - E.g.: the `parallel` construct

- **Example:**
  - Forces all `foo` invokations happen before `bar` invokations
  - Implicit barrier at the end of the parallel region

```
#pragma omp parallel
{
  foo ( ) ;

#pragma omp barrier

  bar ( ) ;
}
```

Exclusive access: `critical` construct

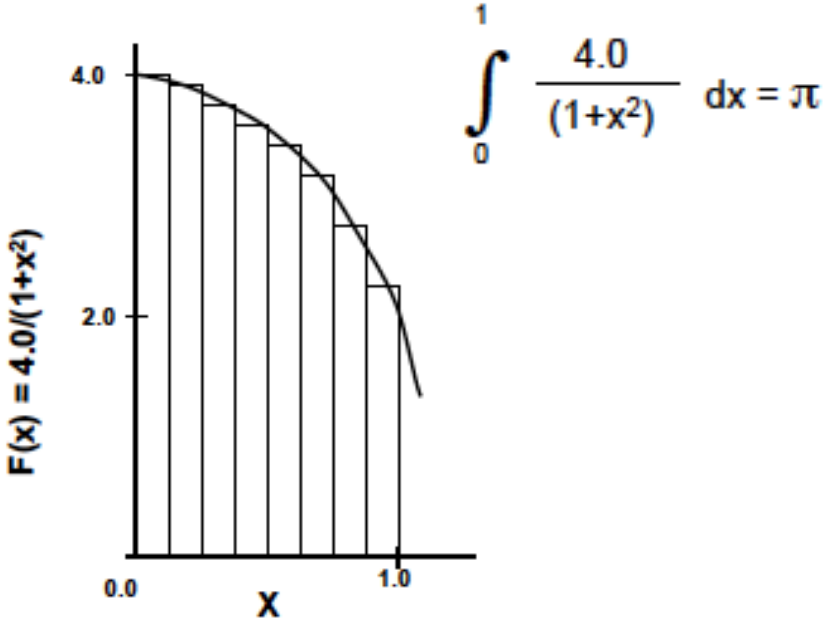
- `#pragma omp critical [(name)]`
  `structured block`
  - Provides a region of mutual exclusion where only one thread can be working at any given time
  - By default all critical regions are the same
  - Multiple mutual exclusion regions by providing them with a `name`
    - ✓ Only those with the same name synchronize

- **Example:**
  - Different names: One thread can update x while another updates y

```
int x=1, y=0;
#pragma omp parallel num_threads ( 4 )
{
#pragma omp critical ( x )
  x++;
#pragma omp critical ( y )
  y++;
}
```

Example: Computation of Pi

- `omp critical` **usage for** `sum` **update**



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

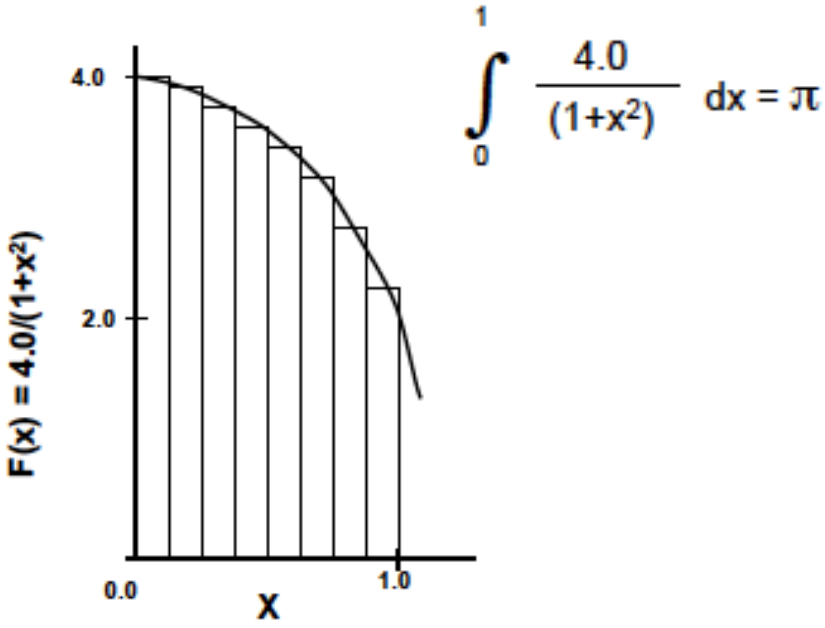
```
static long num steps = 100000;
double step;
#include <omp.h>
#define NUM_THREADS(2)

void main ( )
{
  int i
  double x , pi , sum = 0.0;
  step = 1.0 / (double) num_steps ;
  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(x, i)
  {
    id = omp_get_thread_num( ) ;
    for (i=id+1; i<=num_steps; i=i+NUM THREADS) {
      x = (i-0.5)*step;
#pragma omp critical
      sum = sum + 4.0/(1.0+x*x );
    }
  }
  pi = step*sum;
}
```

Exclusive access: `atomic` construct

- `#pragma omp atomic [ update | read | write ]`
  `expression`
- **Ensures that a specific storage location is accessed atomically, avoiding the possibility of multiple, simultaneous reading and writing threads**
  - Atomic updates: `x += 1, x = x - foo(), x[index[i]]++`
  - Atomic reads: `value = *p`
  - Atomic writes: `*p = value`
- **Only protects the read/operation/write**
- **Usually more efficient than a critical construct**
- **Other clauses and forms for atomic are allowed in the specification**

Example: Computation of Pi

- `omp atomic` **usage for** `sum` **update**



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

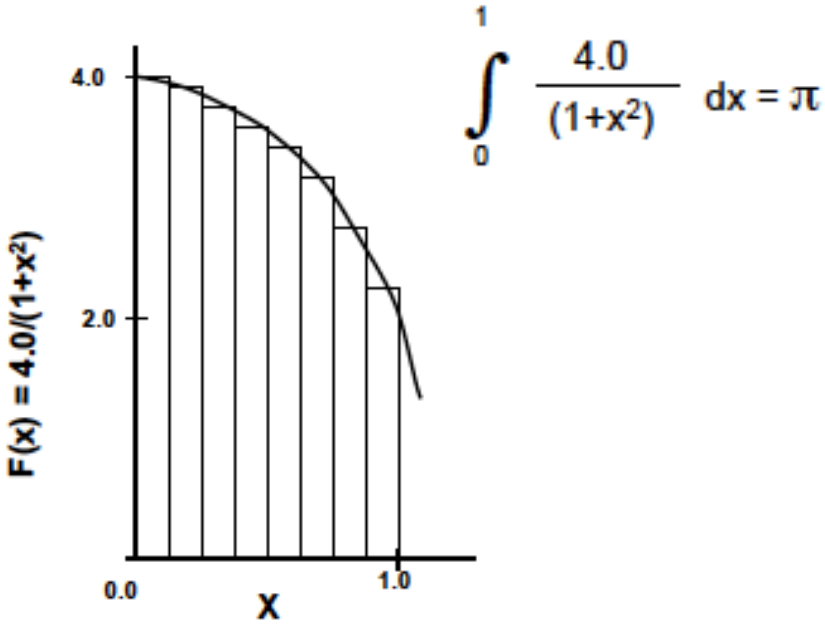
```
static long num steps = 100000;
double step;
#include <omp.h>
#define NUM_THREADS(2)

void main ( )
{
  int i
  double x , pi , sum = 0.0;
  step = 1.0 / (double) num_steps ;
  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(x, i)
  {
    id = omp_get_thread_num( ) ;
    for (i=id+1; i<=num_steps; i=i+NUM THREADS) {
      x = (i-0.5)*step;
#pragma omp atomic
      sum = sum + 4.0/(1.0+x*x );
    }
  }
  pi = step*sum;
}
```

The `reduction` clause

- **Reduction is a very common pattern where all threads accumulate values into a single variable**
  - `reduction ( operator : list )`
    - ✓ Valid operators are: `+, -, *, |, ||, &, &&, ^`
    - ✓ The compiler creates a `private` copy of each variable in list that is properly initialized to the identity value
    - ✓ At the end of the region, the compiler ensures that the `shared` variable is properly (and safely) updated with the partial values of each thread, using the specified operator

Example: Computation of Pi

- `reduction` **usage for** `sum` **update**



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num steps = 100000;
double step;
#include <omp.h>
#define NUM_THREADS(2)

void main ( )
{
  int i
  double x , pi , sum = 0.0;
  step = 1.0 / (double) num_steps ;
  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(x, i) reduction(+:sum)
  {
    id = omp_get_thread_num( ) ;
    for (i=id+1; i<=num_steps; i=i+NUM THREADS) {
      x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x );
     }
  }
  pi = step*sum;
}
```

Exclusive access: Locks

- **OpenMP provides `lock` primitives for low-level synchronization**
  - `omp_init_lock`
    - ✓ Initialize the lock
  - `omp_set_lock`
    - ✓ Acquires the lock
  - `omp_unset_lock`
    - ✓ Releases the lock
  - `omp_test_lock`
    - ✓ Tries to acquire the lock (won't block)
  - `omp_destroy_lock`
    - ✓ Frees lock resources

Exclusive access: Locks

- **Example**

```
#include <omp.h>
void foo ( )
{
omp_lock_t lock;
omp_init_lock(&lock) ;
#pragma omp parallel
{
   omp_set_lock(&lock) ;
   // mutual exclusion region
   omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
}
```

Performance Modeling

- **Outline**
  - Overview
  - Execution Model
  - Threads and Data Access
  - Application Program Interface
  - Thread Synchronization
  - Memory Consistency

Relaxed consistency memory model

- A thread's temporary view of memory is not required to be consistent with memory at all times
- A value written to a variable can remain in the thread’s temporary view until it is forced to memory at a later time
- Likewise, a read from a variable may retrieve the value from. the thread's temporary view, unless it is forced to read from memory

The `flush` construct

- `#pragma omp flush (list)`
  - It enforces consistency between the temporary view and memory for those variables in list
  - Synchronization (implicit or explicit) constructs have an associated `flush` operation

```
#include <omp.h>
void foo ( )
{
  double A[];

#pragma omp parallel
{
  // Variable A is not synchronized
  A[ ... ] = ...;

#pragma omp flush(A)

 // Variable A is synchronized
 // All threads see the same values
 ... = A[ ... ];
}
```

LOOP PARALLELISM IN OPENMP

Loop Level Parallelism

- Work Sharing Concept
- The `Loop` Worksharing
- The `Single` Worksharing

Work Sharing Concept

- **Works-haring constructs divide the execution of a code region among the members of a team**
  - Threads **cooperate** to do some work
  - Better way to split work than using thread-ids
  - Lower overhead than using `tasks` (next section)
    - ✓ But, less flexible
- **In OpenMP, there are four work-sharing constructs:**
  - loop
  - single
  - sections (not that much used, almost substituted by `tasks`. We will skip it)
  - workshare (rarely used, we will skip it)

Loop Work Sharing

- **The `for` construct**
  - `#pragma omp for [ c l a u s e s ]`
    - ✓ `for ( init expr ; test expr ; incr expr )`
- **where some possible clauses are:**
  - `private`
  - `firstprivate`
  - `reduction`
  - `schedule(schedule-kind)`
  - `nowait`
  - `collapse(n)`

Loop Work Sharing

- **The `for` construct**
  - The iterations of the loop(s) associated to the construct are divided among the threads of the team.
    - ✓ Loop iterations must be independent
    - ✓ Loops must follow a form that allows to compute the number of iterations
    - ✓ Valid data types for inductions variables are: integer types, pointers and random access iterators (in C++)
    - ✓ The induction variable(s) are automatically privatized
    - ✓ The default data-sharing attribute is shared
  - It can be merged with the parallel construct:
    - ✓ `#pragma omp parallel for`

Loop Work Sharing

- **Example (Pi computation)**

```
#include <omp.h>
static long num steps = 100000;
double step;
#define NUM_THREADS (2)
void main() {
  int i, id;
  double x, pi, sum;
  step = 1.0 / ( double ) num_steps;
  omp_set_num_threads (NUM_THREADS) ;
#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 = sum * step;
}
```

The `schedule` clause

- **The `schedule` clause determines which iterations are executed by each thread.**
  - If no `schedule` clause is present then is implementation-defined
- **There are several possible options as schedule:**
  - `static[,chunk]`
  - `dynamic[,chunk]`
  - `guided[,chunk]`

The `schedule` clause

- **`static`**
  - The iteration space is broken in **chunks** of approximately size `N/num threads`. Then these chunks are assigned to the threads in a Round-Robin fashion.
- **`static,N` (also called interleaved)**
  - The iteration space is broken in **chunks** of size N. Then these chunks are assigned to the threads in a Round-Robin fashion.
- **Characteristics of static schedules**
  - Low overhead
  - Good locality (usually)
  - Can have load imbalance problems

Exercise: check on openmp.org the exact arithmetic to compute the iteration distribution!!!

The `schedule` clause

- **`dynamic,N`**
  - Threads dynamically grab chunks of N iterations until all iterations have been executed. If no chunk is specified, N = 1.
- **`guided,N`**
  - Variant of dynamic. The size of the chunks decreases as the threads grab iterations, but it is at least of size N. If no chunk is specified, N = 1.
- **Characteristics of dynamic schedules**
  - Higher overhead
  - Not very good locality (usually)
  - Can solve imbalance problems

Exercise: check on openmp.org the description of how iterations are distributed!!!

The `nowait` clause

- When a worksharing has a `nowait` clause then the implicit barrier at the end of the loop is removed.
- This allows to overlap the execution of non-dependent loops/tasks/worksharings

```
#pragma omp for nowait
  for (i = 0 ; i < n ; i++)
    v[i] = 0 ;

#pragma omp for
 for (i = 0 ; i < n ; i++)
   a[i] = 0 ;
```

The `nowait` clause

- **Useful to overlap the execution of two (or more) consecutive loops if they have the same static schedule and all have the same number of iterations.**

```
#pragma omp for schedule (static, 2) nowait
  for (i = 0 ; i < n ; i++)
    v[i] = ...;

#pragma omp for schedule (static, 2)
  for (i = 0 ; i < n ; i++)
    a[i] = v[i] * v[i] ;
```

The collapse clause

- **Allows to distribute work from a set of `n` nested loops.**
  - Loops must be perfectly nested
  - The nest must traverse a rectangular iteration space

```
#pragma omp for collapse(2)
  for (i = 0 ; i < N; i++)
    for (j = 0 ; j < M; j++)
      foo(i, j) ;
```

i-loop and j-loop are folded and iterations distributed among all threads.
Both i and j are privatized

The `single` construct

- **Assigns work to a single thread**

```
#pragma omp single [ c l a u s e s ]
    structured block
```

  - Clauses can be:
    - ✓ `private`
    - ✓ `firstprivate`
    - ✓ `nowait`
  - Only one thread of the team executes the structured block
  - There is an implicit barrier at the end

TASK PARALLELISM IN OPENMP

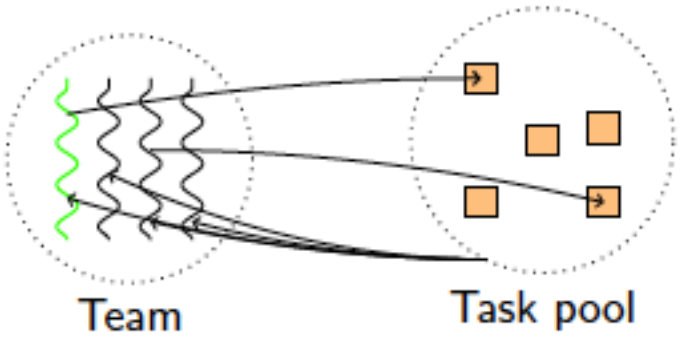
OpenMP Tasking Model

- OpenMP Tasks

- Task Synchronization

Task parallelism model

- **Tasks are work units whose execution may be deferred**
  - In some cases, they can also be executed immediately
- **Threads of the team cooperate to execute them**
- **We distinguish between two essential events**
  - `Task Creation`
  - `Task Execution`



Task Creation: Implicit and Explicit

- **Creating tasks**
  - Implicit
    - ✓ Parallel regions create tasks, as many as threads execution in the team
    - ✓ One `implicit` task is created and assigned to each thread
  - Explicit
    - ✓ Each thread that encounters a task construct
      - – Packages the code and data
      - – Creates a new `explicit` task

Creating (explicit) tasks

- **The `task` construct**

```
#pragma omp task [ c l a u s e s ]
    structured block
```

- **Where some possible clauses are:**
  - `shared`
  - `private`
  - `firstprivate`
    - ✓ Values are captured at creation time
  - `if(expression)`
  - `final(expression)`
  - `mergeable`

Example: List Traversal

- **Code scheme**
  - **We need threads to execute the tasks ...**
  - **... but not that many! This will generate multiple traversals**

```
List L;

  L = ...;

  ...
#pragma omp parallel
  traverse_list(L);

  ...


void traverse_list (List* L)
{
  Element E;
  for ( E = L->first; E ; E = E->>next) {
#pragma omp task
    process(E);
  }
}
```

Example: List Traversal

- **Code scheme**
  - Using `single`
    - ✓ **One** thread **creates** the tasks of the traversal. **The rest** (and this one, once task generation is finished) **cooperate** to execute them

```
  ...
#pragma omp parallel
{
#pragma omp single
  traverse_list(L);
}
  ...


void traverse_list (List* L)
{
  Element E;
  for ( E = L->first; E ; E = E->>next) {
#pragma omp task
    process(E);
  }
}
```

Default task data-sharing attributes

- **When no data clauses are specified, some rules apply:**
  - Global variables are `shared`
  - Variables declared in the scope of a task are `private`
  - The rest are `firstprivate` except when a `shared` attribute can be lexically inherited

Default task data-sharing attributes

- **Exercise**
  - Global variables are `shared`
  - Variables declared in the scope of a task are `private`
  - The rest are `firstprivate` except when a `shared` attribute can be lexically inherited

- **Explain the scope of the variables**

```
int a;
void foo( ) {
  int b, c;
#pragma omp parallel shared (b)
#pragma omp parallel private (b)
{
  int d;
#pragma omp task
  {
    int e;
    a = // shared
    b = // firstprivate
    c = // shared
    d = // firstprivate
    e = // private
  }
}}
```

Immediate task execution

- **The `if` clause**
  - If the `expression` of an if clause evaluates to **`false`**
    - ✓ The encountering task is suspended
    - ✓ The new task is **executed immediately**
      - – with its own data environment
      - – different task with respect to synchronization
    - ✓ The parent task resumes when the task finishes
    - ✓ Allows implementations to **optimize** task creation

Immediate task execution (nested)

- **The `final` and `mergeable` clauses**
  - `final(expression)`
    - ✓ If the `expression` of an if clause evaluates to **`true`**
      - The generated task and all of its child tasks will be **`final`**
    - ✓ The execution of a `final` task is sequentially included in the generating task (executed immediately)
  - `mergeable`
    - ✓ When a `mergeable` clause is present on a task construct, and the generated task is an **included** task, the implementation may generate a **merged task** instead (i.e. no task and context creation for it).

Immediate task execution (nested)

- `final` **and** `mergeable` **tasks (data race!)**

```
int fibonacci(int n) {

  int i, j;

  if (n<2) return n;

#pragma omp task shared(i) final(n <= THOLD) mergeable
  i = fibonacci(n-1);

#pragma omp task shared(j) final(n <= THOLD) mergeable
  j = fibonacci (n-2);

  ...
  return (i+j);
}
```

OpenMP Tasking Model

- **OpenMP Tasks**

- **Task Synchronization**

Task Synchronization

- **There are two types of task barriers:**
  - `taskwait`
    - ✓ Suspends the current task waiting on the completion of **child tasks** of the current task. The taskwait construct is a stand-alone directive
  - `taskgroup`
    - ✓ Suspends the current task at the end of structured block waiting on completion of **child tasks** of the current task and their **descendent tasks**.

Task Synchronization: `taskwait`

```
#pragma omp taskwait
```

```
#pragma omp task {}          // T1

#pragma omp task             // T2
{
    #pragma omp task {}      // T3
}

#pragma omp task {}          // T4

#pragma omp taskwait
```

Only T1, T2 and T4 are guaranteed to have finished here!!!

Taskwait for correct Fibonacci parallelization

```
#pragma omp taskwait
```

```
int fibonacci(int n) {

  int i, j;

  if (n<2) return n;

#pragma omp task shared(i) final(n <= THOLD) mergeable
  i = fibonacci(n-1);

#pragma omp task shared(j) final(n <= THOLD) mergeable
  j = fibonacci (n-2);

#pragma omp taskwait

  return (i+j);
}
```

Task Synchronization: `taskgroup`

```
#pragma omp taskgroup
      structured block

    #pragma omp task {}          // T1
    #pragma omp taskgroup
    {

    #pragma omp task             // T2
    {
        #pragma omp task {}      // T3
    }

    #pragma omp task {}          // T4

    } // taskgroup
```

Only T2, T3 and T4 are guaranteed to have finished here!!!

Data sharing inside tasks

- **In addition one can use `critical` and `atomic` to synchronize the access to `shared` data inside the `task`**

```
void process ( Element e )
{
. . .
#pragma omp atomic
  solutions _found++;
. . .
}
```

Task dependences

- **Definition of dependences between sibling tasks (i.e. from the same father)**

```
#pragma omp task  [ depend ( in : varlist) ]
                  [ depend ( out : varlist) ]
                  [ depend ( inout : varlist) ]
```

- **Task dependences are derived from the dependence type (`in`, `out` or `inout`) and its items in `varlist`. This list may include array sections**

Task dependences

- The `in` dependence-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` dependence-type list
- The `out` and `inout` dependence-types: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` dependence-type list.

Task dependences

- **Example: wave-front execution**

```
#pragma omp parallel private(i ,j)
#pragma omp single
{
  for (i=1; i<n i++) {
    for (j=1; j<n ; j++) {
      #pragma omp task depend(in : block[i-1][j], block[i][j-1])
                       depend(out : block[i][j])
      foo(i, j);
    }
  }
}
```

Reduction operations and tasks

- **Exercise:**
  - Check current specification in openmp.org