Hans J. Johnson

# OpenMP Introduction

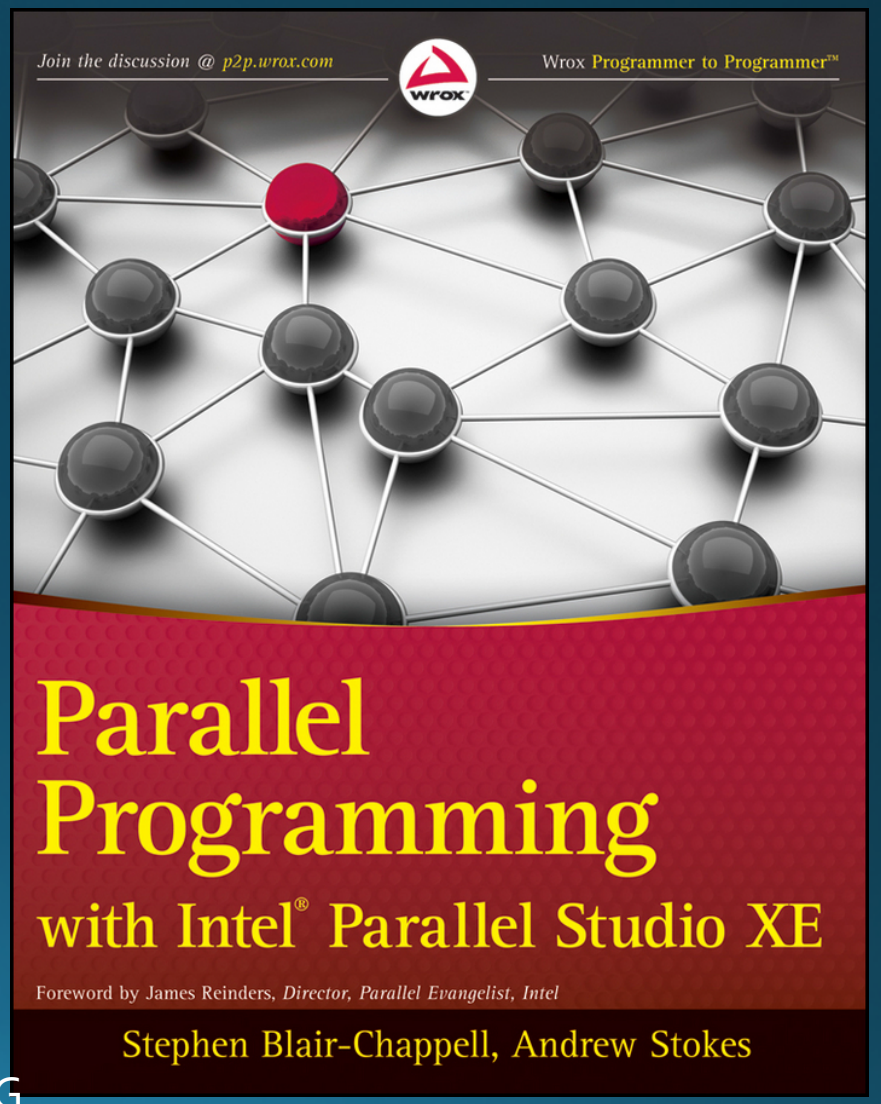
# Slides From Many Sources

OpenMP

## A "Hands-on" Introduction to OpenMP*


Tim Mattson
Principal Engineer
Intel Corporation
timothy.g.mattson@intel.com

Larry Meadows
Principal Engineer
Intel Corporation
lawrence.f.meadows@intel.com

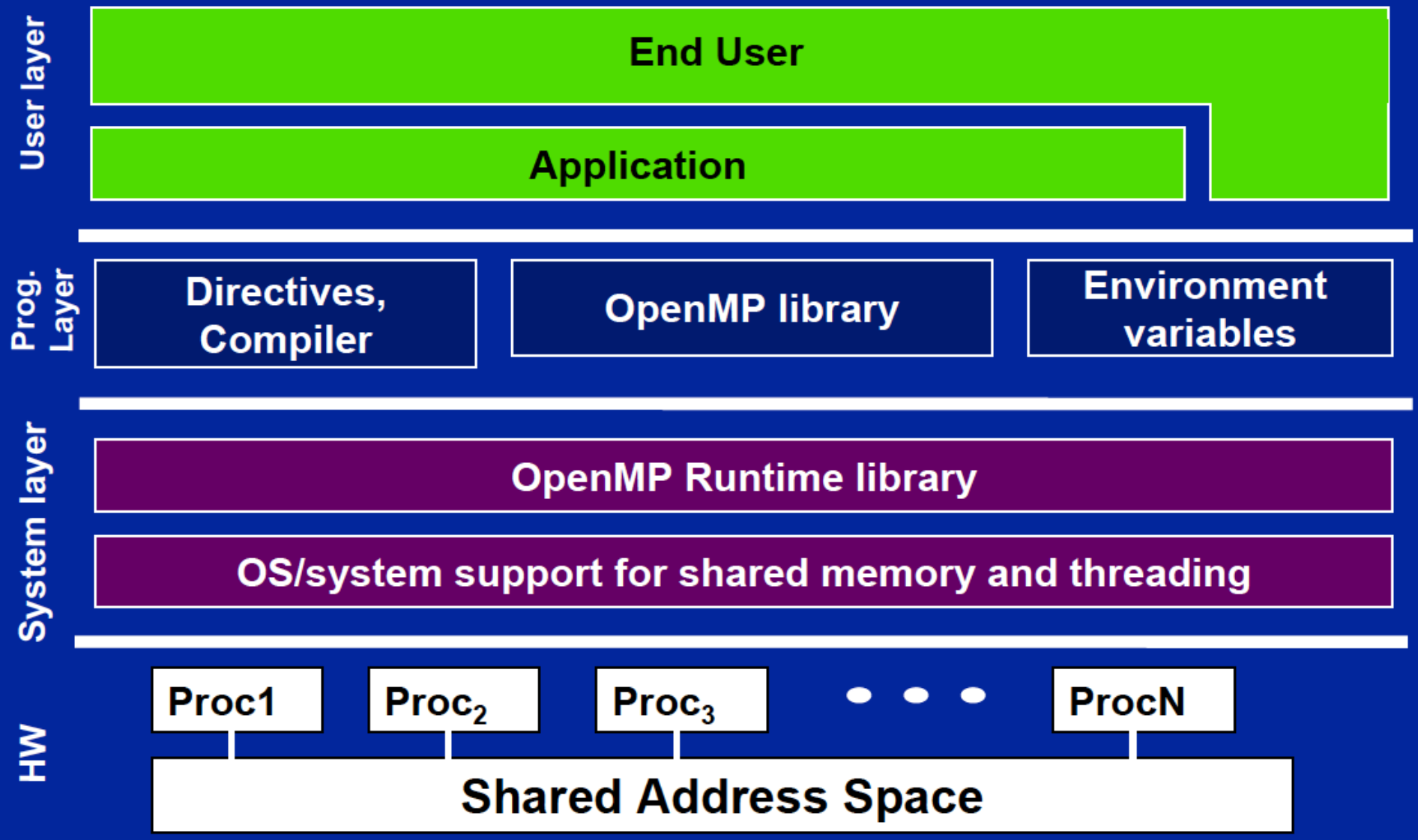




https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG

# OpenMP:
## An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

# OpenMP Basic Defs: Solution Stack



# OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp *construct [clause [clause]…]*

  - Example

    *#pragma omp parallel num_threads(4)*

- Function prototypes and types in the file:

  #include <omp.h>

- Most OpenMP* constructs apply to a "structured block".
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It's OK to have an exit() within the structured block.

# Exercise 1, Part A: Hello world

## Verify that your environment works

- **Write a program that prints “hello world”.**

```
void main()
{



    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```

# In the HelloWorld Directory For Example Code

```
CXX = icpc
## -std=c++11: Enables c++11 conformance
## -O2: optimize for speed
## -xHost tune code to only run on this host archetecture
## -ipo: enable interprocedural optimization
## -g: generate debug information
## -inline-level=1: disables inlining (=0 disables all inlineing)
## -openmp: enables openmp directives
CXXFLAGS = -std=c++11 -O2 -xHOST -ipo -g -inline-level=1 -openmp

all: HelloWorld.exe

%.o: ../src/%.cpp
	$(CXX) -c -o $@ $< $(CXXFLAGS)

HelloWorld.exe: HelloWorld.o
	$(CXX) -o $@ $^ $(CXXFLAGS) $(LIBS)
	echo "========== DOING $@"
	./$@

.PHONY: clean
clean:
	rm -f *.o *~ core $(INCDIR)/*~ log_run*
```

# Exercise 1, Part B: Hello world

## Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

```
#include “omp.h”
void main()
{
 #pragma omp parallel
 {
    int ID = 0;
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
 }
}
```

**Switches for compiling and linking**

| | |
|---|---|
| **-fopenmp** | **gcc** |
| **-mp** | **pgi** |
| **/Qopenmp** | **intel** |

# In the HelloWorld Directory For Example Code

Step 1:
Build and run
Step 2:
Remove
Build and run
Step 3:
Remove
Build and run

```
#include <stdlib.h>
#include <iostream>
#include <omp.h>

int main()
{
// Add first
//#pragma omp parallel default(none) shared(std::cout)
  {
// Add second
//#pragma omp critical
   {
   int id = omp_get_thread_num();
   std::cout << "HelloWorld(" << id << ")"
             << std::flush << std::endl;
   }
  }
  return EXIT_SUCCESS;
}
```

# Exercise 1: Solution
## A multi-threaded "Hello world" program

- Write a multithreaded program where each thread prints "hello world".

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

OpenMP include file

number of threads

End of the Parallel region

Runtime library function to return a thread ID.

**Sample Output:**

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)

# OpenMP Overview:
## How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

# Runtime Library routines

- Runtime environment routines:
  - Modify/Check the number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Are we in an active parallel region?
    - omp_in_parallel()
  - Do you want the system to dynamically vary the number of threads from one parallel construct to another?
    - omp_set_dynamic, omp_get_dynamic();
  - How many processors in the system?
    - omp_num_procs()

**…plus a few less commonly used routines.**

# Runtime Library routines

- **To use a known, fixed number of threads in a program, (1) tell the system that you don't want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.**

```
#include <omp.h>
void main()
{   int num_threads;
     omp_set_dynamic( 0 );
     omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
   {    int id=omp_get_thread_num();
#pragma omp single
          num_threads = omp_get_num_threads();
        do_lots_of_stuff(id);
    }
}
```

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.

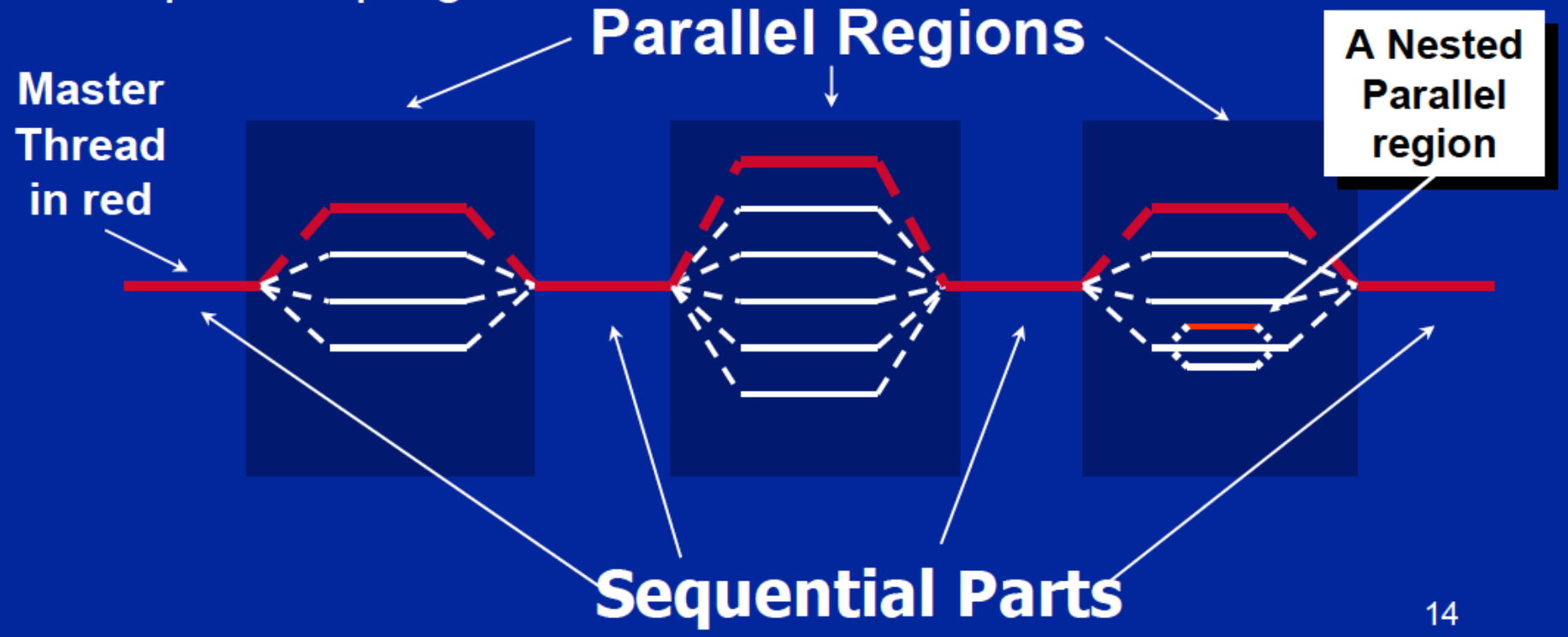
Protect this op since Memory stores are not atomic

**Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.**

# OpenMP Programming Model:

## Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



# A brief detour.

- Before we advance our knowledge of OpenMP, we need to think about anonymous codeblocks that are run many times.

# LambdaExampleCode: NoLambda

```
size_t global_var_i; //
static void global_set_values(float & in)
{
  in = global_var_i;
  global_var_i++;
}

float global_slope;
float global_intercept;
static void global_print_line(float & in)
{
  in=in*global_slope+global_intercept;
}

static void print( const float in )
{
  std::cout << in << " ";
}
```

```
int main()
{
  constexpr size_t VECTOR_SIZE=100;
  float x[VECTOR_SIZE];

  global_var_i=0;
  std::for_each(x, x+VECTOR_SIZE,
                global_set_values );

  global_slope=2.0;
  global_intercept=10.0;
  std::for_each(x, x+VECTOR_SIZE,
                global_print_line );

  std::for_each(x, x+VECTOR_SIZE,
                print );
  std::cout << std::endl;
  return EXIT_SUCCESS;
}
```

# What's Wrong with Previous Code

```
size_t global_var_i; //
static void global_set_values(float & in)
{
  in = global_var_i;
  global_var_i++;
}

float global_slope;
float global_intercept;
static void global_print_line(float & in)
{
  in=in*global_slope+global_intercept;
}

static void print( const float in )
{
  std::cout << in << " ";
}
```

- It smell's bad!
  - Global variables are bad.
  - Functions need to be created to do simple task.
- Would it be great if the "work package" could be bundled nicer?
  - Work package is nearly trivial
  - Work package is only needed in the 3rd argument of the for_each loop.
- LAMBDA FUNCTIONS!
  - We will investigate the C++ concepts for lambda functions/functors/anonymous functions as a gateway to understanding OpenMP concepts.

# Lambda Functions
(a.k.a functors, anonymous functions)

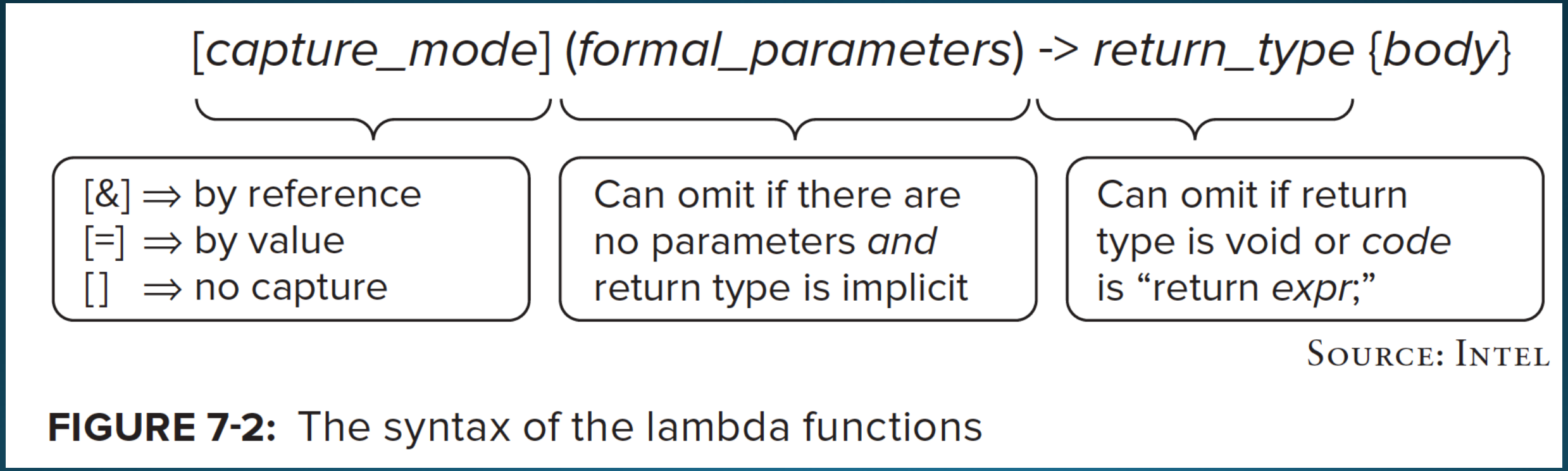
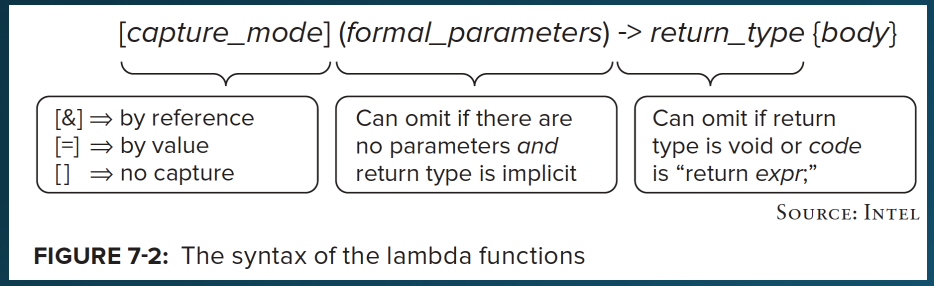



**FIGURE 7-2:** The syntax of the lambda functions

# Complete Solution Using Lambdas

## 02Lambda.cpp




SOURCE: INTEL

**FIGURE 7-2:** The syntax of the lambda functions

```
int main()
{
  constexpr size_t VECTOR_SIZE=100;
  float x[VECTOR_SIZE];

  size_t i=0;
  std::for_each(x, x+VECTOR_SIZE,
       [&i] (float &in) ->void { in = i; ++i; } );

  // Solve x=m*x+b
  const float m=2.0F;
  const float b=10.0F;
  std::for_each(x, x+VECTOR_SIZE,
        [m,b](float & in) -> void { in = m*in+b; } );

  std::for_each(x, x+VECTOR_SIZE,
        [m,b](float in) -> void { std::cout <<  in << " "; } );

  std::cout << std::endl;
  return EXIT_SUCCESS;
}
```

```
/* OLD CODE WITH GLOBALS AND FUNCTIONS */
std::for_each(x, x+VECTOR_SIZE, global_set_values );
global_i=0;
^^^^^^^^^^^^
aaaaaaaaaaaa

void global_set_values(float &in) { in = global_i; global_i++; }
^^^^                   ^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
cccc                   bbbbbbbbb  ddddddddddddddddddddddddddddd
*/
size_t i=0;
std::for_each(x, x+VECTOR_SIZE,
      [&i] (float &in) ->void { in = i; ++i; } );
//    ^^^^ ^^^^^^^^^^^^ ^^^^^^ ^^^^^^^^^^^^^^^^
//    aaaa bbbbbbbbbbb  ccccccc ddddddddddddddd
//
//    a - capture section,
//        takes variables from calling environment
//        (like a global variable)
//    b - function arguments - just like normal functions
//    c - The return type of the lambda function
//    d - lambda function body.
```

# 03Lambda_SeparateTasks.cpp

- Now let's Identify ways to break up the work into tasks

```
constexpr size_t MAXTHREADS=4;
constexpr size_t CHUNKSIZE=VECTOR_SIZE/MAXTHREADS;

size_t tid;        //The thread number
size_t startIndex; //Processing start point
size_t stopIndex;  //Processing end point

//Now process as separate tasks in serial
 {
   {
   // TASK#0
   tid=0;
   startIndex=tid*(CHUNKSIZE);
   stopIndex=( tid == ( MAXTHREADS-1) ) ? VECTOR_SIZE : (tid+1)*(CHUNKSIZE);
   std::for_each(x+startIndex, x+stopIndex,
     [m,b,tid](float &in) -> void { in = m*in+b ; } );
   }
   {
   // TASK#1
   tid=1;
   startIndex=tid*(CHUNKSIZE);
```

# Sections worksharing Construct

- The *Sections* worksharing construct gives a different structured block to each thread.

```
#pragma omp parallel
{
  #pragma omp sections
  {
  #pragma omp section
        X_calculation();
  #pragma omp section
        y_calculation();
  #pragma omp section
        z_calculation();
  }
}
```

**By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.**

# 04Lambda_MultiProcessor.cpp

- Parallelized Version!

```
    constexpr size_t MAXTHREADS=4;
    constexpr size_t CHUNKSIZE=VECTOR_SIZE/MAXTHREADS;

    size_t tid;        //The thread number
    size_t startIndex; //Processing start point
    size_t stopIndex;  //Processing end point

std::cout << "Only using " << MAXTHREADS << " of " << omp_get_max_threads() << std::endl;

#pragma omp parallel sections default(none) num_threads(MAXTHREADS) private(tid,startIndex,stopIndex) fir
stprivate(m,b) shared(x)
    {
#pragma omp section
      {
      // TASK#0
      tid=0;
      startIndex=tid*(CHUNKSIZE);
      stopIndex=( tid == ( MAXTHREADS-1) ) ? VECTOR_SIZE : (tid+1)*(CHUNKSIZE);
      std::for_each(x+startIndex, x+stopIndex,
        [m,b,tid](float &in) -> void { in = m*in+b ; } );
      }
#pragma omp section
      {
      // TASK#1
      tid=1;
      startIndex=tid*(CHUNKSIZE);
```

# Data environment:
## Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

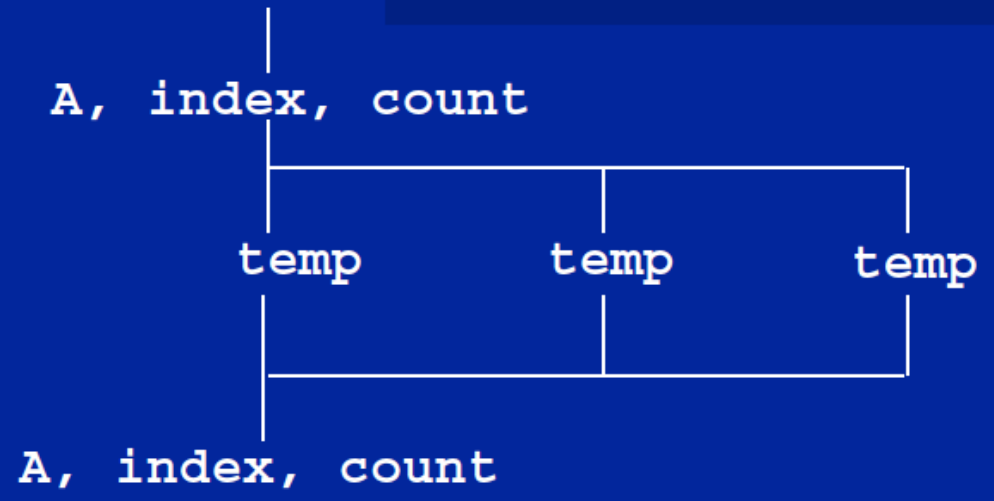
# Data sharing: Examples

```
double A[10];
int main() {
int index[10];
#pragma omp parallel
      work(index);
printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

**A, index and count are shared by all threads.**

**temp is local to each thread**



# Data sharing:
## Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - LASTPRIVATE
- The default attributes can be overridden with:
  - DEFAULT (PRIVATE | SHARED | NONE)

    DEFAULT(PRIVATE) is Fortran only

All data clauses apply to parallel constructs and worksharing constructs except "shared" which only applies to parallel constructs.

# Data Sharing: Private Clause

- private(var) creates a new local copy of var for each thread.
  - The value is uninitialized
  - In OpenMP 2.5 the value of the shared variable is undefined after the region

```
void wrong() {
      int tmp = 0;
#pragma omp for private(tmp)
      for (int j = 0; j < 1000; ++j)
                  tmp += j;
      printf("%d\n", tmp);
}
```

tmp was not initialized

tmp: 0 in 3.0, unspecified in 2.5

# Data Sharing: Firstprivate Clause

- **Firstprivate is a special case of private.**
  - **Initializes each private copy with the corresponding value from the master thread.**

```
void useless() {
    int tmp = 0;
#pragma omp for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j)
      tmp += j;
    printf("%d\n", tmp);
 }
```

tmp with an initial value of 0

tmp: 0 in 3.0, unspecified in 2.5

# Data sharing: Lastprivate Clause

- **Lastprivate passes the value of a private from the last iteration to a global variable.**

```
void closer() {
    int tmp = 0;
#pragma omp parallel for firstprivate(tmp) \
  lastprivate(tmp)
  for (int j = 0; j < 1000; ++j)
     tmp += j;
  printf("%d\n", tmp);
}
```

Each thread gets its own tmp an initial value of 0

tmp is defined as its value at the "last sequential" iteration (i.e., for j=999)

# Data Sharing:
## A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```
variables A,B, and C = 1
#pragma omp parallel private(B) firstprivate(C)
```

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

**Inside this parallel region ...**

- "A" is shared by all threads; equals 1
- "B" and "C" are local to each thread.
  - B's initial value is undefined
  - C's initial value equals 1

**Outside this parallel region ...**

- The values of "B" and "C" are unspecified in OpenMP 2.5, and in OpenMP 3.0 if referenced in the region but outside the construct.

# Data Sharing: Default Clause

- Note that the default storage attribute is DEFAULT(SHARED) (so no need to use it)
  - Exception: #pragma omp task
- To change default: DEFAULT(PRIVATE)
  - *each* variable in the construct is made private as if specified in a private clause
  - mostly saves typing
- DEFAULT(NONE): *no* default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

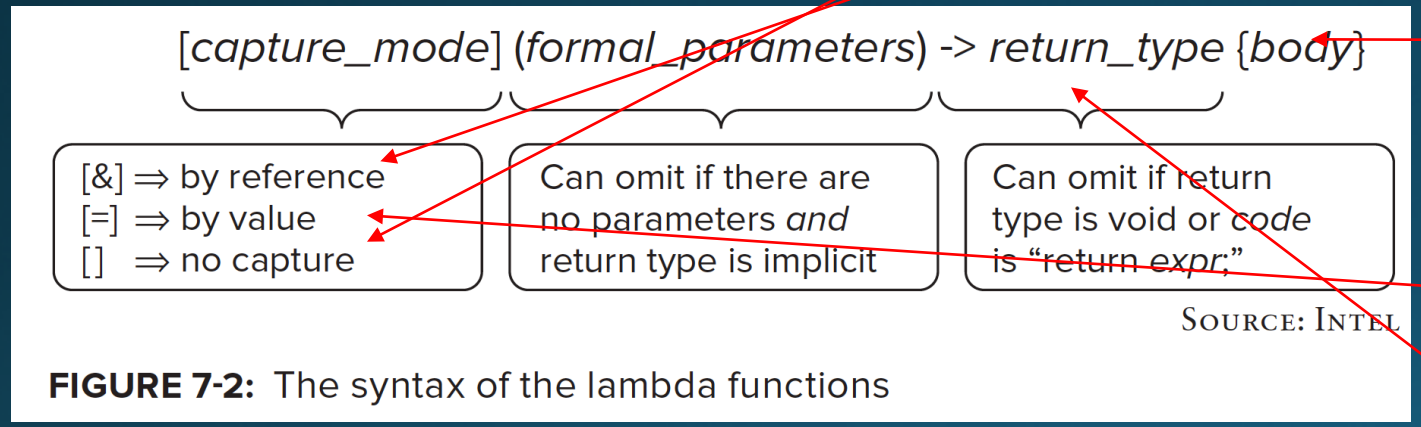



**FIGURE 7-2:** The syntax of the lambda functions

C++11 Lambda Function Invocation

**default(shared | none)**
Explicitly determines the default data-sharing attributes of variables that are referenced in a **parallel**, **task**, or **teams** construct, causing all variables referenced in the construct that have implicitly determined data-sharing attributes to be shared.

**shared**(*list*)
Declares one or more list items to be shared by tasks generated by a **parallel**, **task**, or **teams** construct. The programmer must ensure that storage shared by an explicit **task** region does not reach the end of its lifetime before the explicit task region completes its execution.

**private**(*list*)
Declares one or more list items to be private to a task or a SIMD lane. Each task that references a list item that appears in a **private** clause in any statement in the construct receives a new list item.

**firstprivate**(*list*)
Declares list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

**lastprivate**(*list*)
Declares one or more list items to be private to an implicit task or to a SIMD lane, and causes the corresponding original list item to be updated after the end of the region.

OpenMP Code Block Invocation

# Loop worksharing Constructs

## A motivating example

**Sequential code**

```
for(i=0;I<N;i++)  { a[i] = a[i] + b[i];}
```

**OpenMP parallel region**

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;I<iend;i++)  { a[i] = a[i] + b[i];}
}
```

**OpenMP parallel region and a worksharing for construct**

```
#pragma omp parallel
#pragma omp for
        for(i=0;I<N;i++)  { a[i] = a[i] + b[i];}
```

# 05Lambda_ManualParallel.cpp

- Parallelized Version!

```
#pragma omp parallel default(none) num_threads(MAXTHREADS) private(tid,startIndex,
stopIndex) firstprivate(m,b) shared(x)
    {
      {
      // TASK#0
      tid=omp_get_thread_num();
      startIndex=tid*(CHUNKSIZE);
      stopIndex=( tid == ( MAXTHREADS-1) ) ? VECTOR_SIZE : (tid+1)*(CHUNKSIZE);
      std::for_each(x+startIndex, x+stopIndex,
        [m,b,tid](float &in) -> void { in = m*in+b ; } );
      }
    }
```

# o6Lambda_ParallelFor.cpp

```cpp
int main()
{
  constexpr size_t VECTOR_SIZE=100;
  float x[VECTOR_SIZE];

  size_t i=0;
  std::for_each(x, x+VECTOR_SIZE,
       [&i] (float &in) ->void { in = i; ++i; } );

  // Solve x=m*x+b
  const float m=2.0F;
  const float b=10.0F;
  //size_t tid;         //The thread number
  //size_t startIndex; //Processing start point
  //size_t stopIndex;  //Processing end point

#pragma omp parallel for default(none) firstprivate(m,b) shared(x)
  for(size_t i=0; i< VECTOR_SIZE; ++i) {           x[i]=m*x[i]+b;}

  std::cout << "\n\n" << std::endl;
  // Now print
  std::for_each(x, x+VECTOR_SIZE,
        [m,b](const float in) -> void { std::cout <<  in << " "; } );

  std::cout << std::endl;
  return EXIT_SUCCESS;
}
```

NOTE: The utility variables are no longer needed

# o6Lambda_ParallelFor.cpp

- Parallelized Version!

```
std::for_each(x+0,x+VECTOR_SIZE, [m,b](float &in)->void{in=m*in+b;});
```

```
#pragma omp parallel for default(none) firstprivate(m,b) shared(x)
  for(size_t i=0; i< VECTOR_SIZE; ++i) {              x[i]=m*x[i]+b;}
```

# loop worksharing constructs:
## The schedule clause

- **The schedule clause affects how loop iterations are mapped onto threads**
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0).

# loop work-sharing constructs:
## The schedule clause

| Schedule Clause | When To Use |
|---|---|
| STATIC | Pre-determined and predictable by the programmer |
| DYNAMIC | Unpredictable, highly variable work per iteration |
| GUIDED | Special case of dynamic to reduce scheduling overhead |

Least work at scheduling compile-time

runtime : complex logic used at

OpenMP 3.0

# Nested parallelism

- Better support for nested parallelism
- Per-thread internal control variables
  - Allows, for example, calling `omp_set_num_threads()` inside a parallel region.
  - Controls the team sizes for next level of parallelism
- Library routines to determine depth of nesting, IDs of parent/grandparent etc. threads, team sizes of parent/grandparent etc. teams

```
omp_get_active_level()
omp_get_ancestor(level)
omp_get_teamsize(level)
```

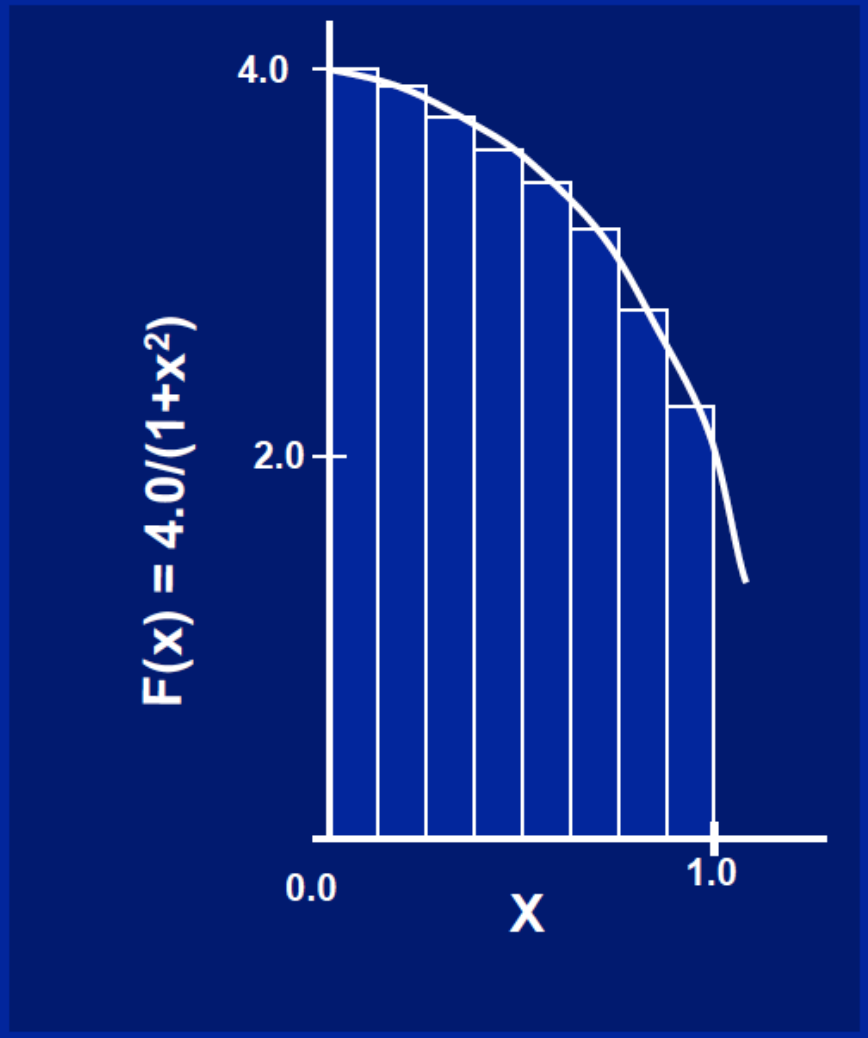
# Loops (cont.)

- Allow collapsing of perfectly nested loops

```
!$omp parallel do collapse(2)
do i=1,n
     do j=1,n
          .....
     end do
end do
```

- Will form a single loop and then parallelize that

# Exercises 2 to 4:
## Numerical Integration




Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width $\Delta x$ and height $F(x_i)$ at the middle of interval i.