Parallel Programming with OPENMP

Akhila Prabhakaran

Consultant Scientist/Technologist

Supercomputer Education and Research Centre

Indian Institute of Science

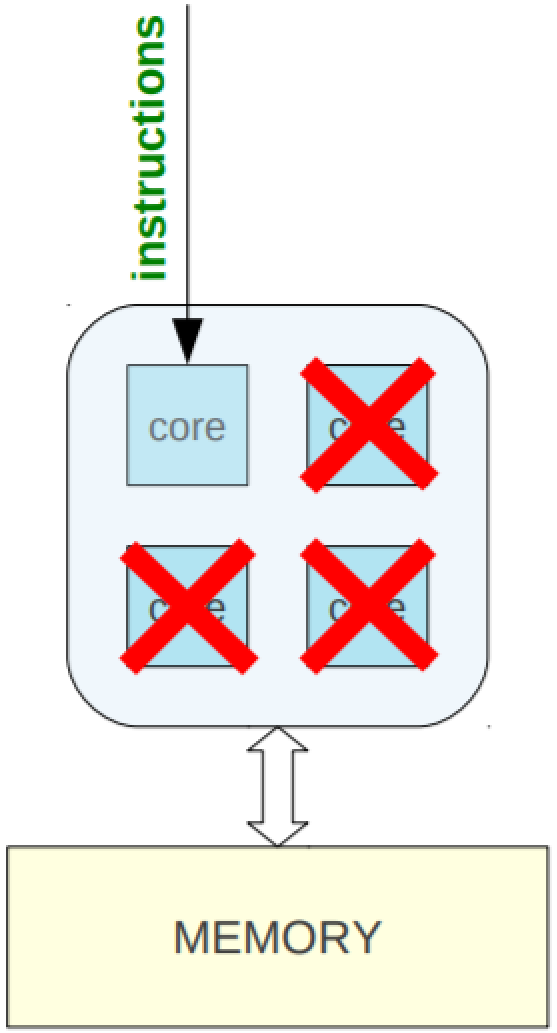
akhilap@iisc.ac.in

OPENMP: Motivation

When you run sequential program

- Instructions executed on 1 core
- Other cores are idle

Waste of available resources. We want all cores to be used to execute program.



OPENMP Motivation

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
//Do this part in parallel

  printf(“Hello World”);

  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main()
{
 omp_set_num_threads(16);
//Do this part in parallel
 #pragma omp parallel
 {
   //structured block of code
   printf(“Hello World”);
 }
  return 0;
}
```

OPENMP : Overview

- Collection of compiler directives and library functions for creating parallel programs for shared-memory computers.
- The "MP" in OpenMP stands for "multi-processing"(shared-memory parallel computing)
- Combined with C, C++, or Fortran to create a multithreading programming language, in which all processes are assumed to share a single address space.
- Based on the fork / join programming model: all programs start as a single (master) thread, fork additional threads where parallelism is desired (the parallel region), then join back together.
- Version 1.0 with fortran in 1997, supporting C & C++ there after, currently at version 5.0 in 2018.

OpenMP: Goals

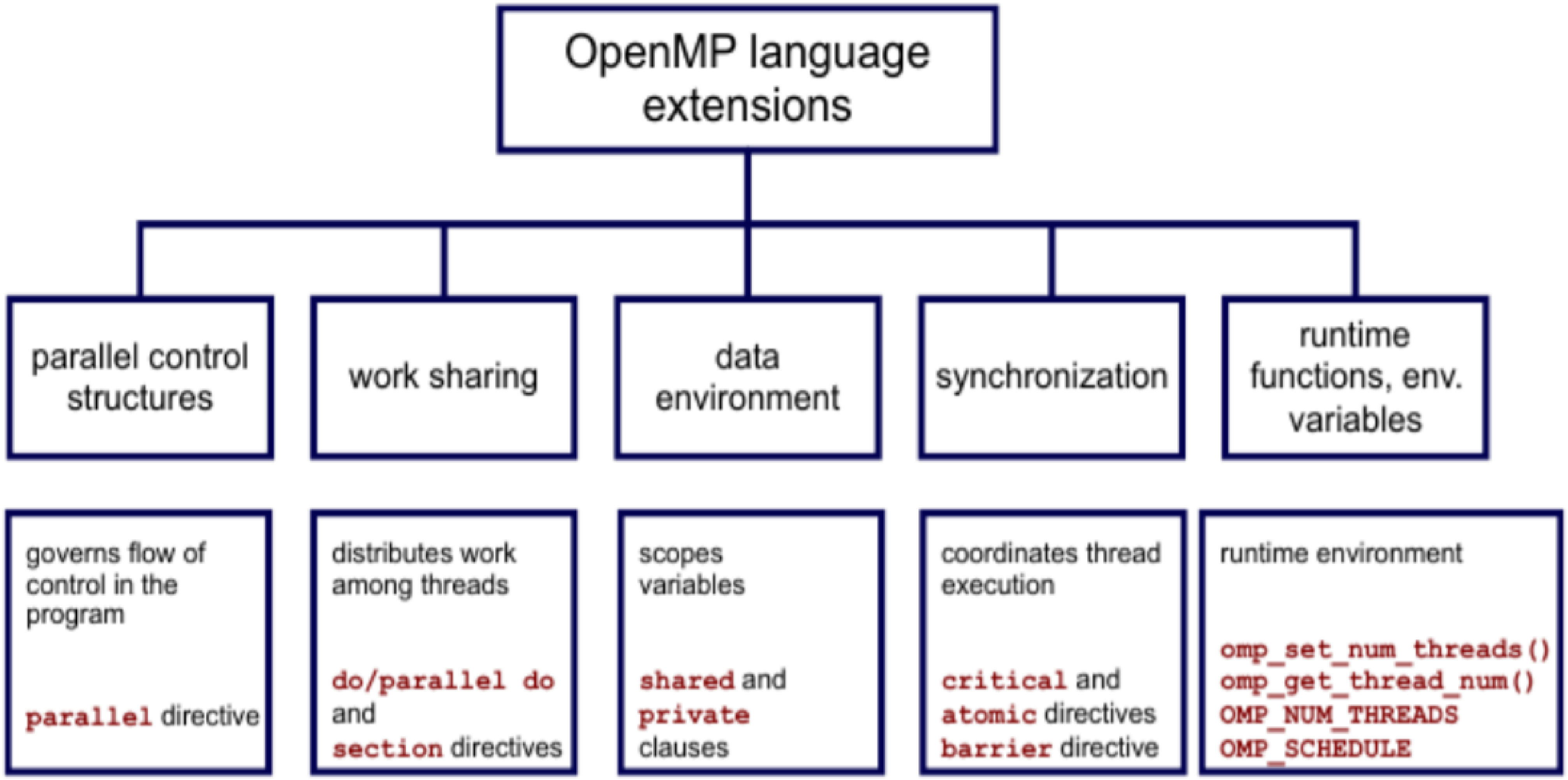
Standardization: Provide a standard among a variety of shared memory architectures/platforms

Lean and Mean: Establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.

Ease of Use: Provide capability to incrementally parallelize a serial program. Provide the capability to implement both coarse-grain and fine-grain parallelism

Portability: Supports Fortran (77, 90, 95…), C, and C++. Public forum for API and membership

OpenMP: Core Elements

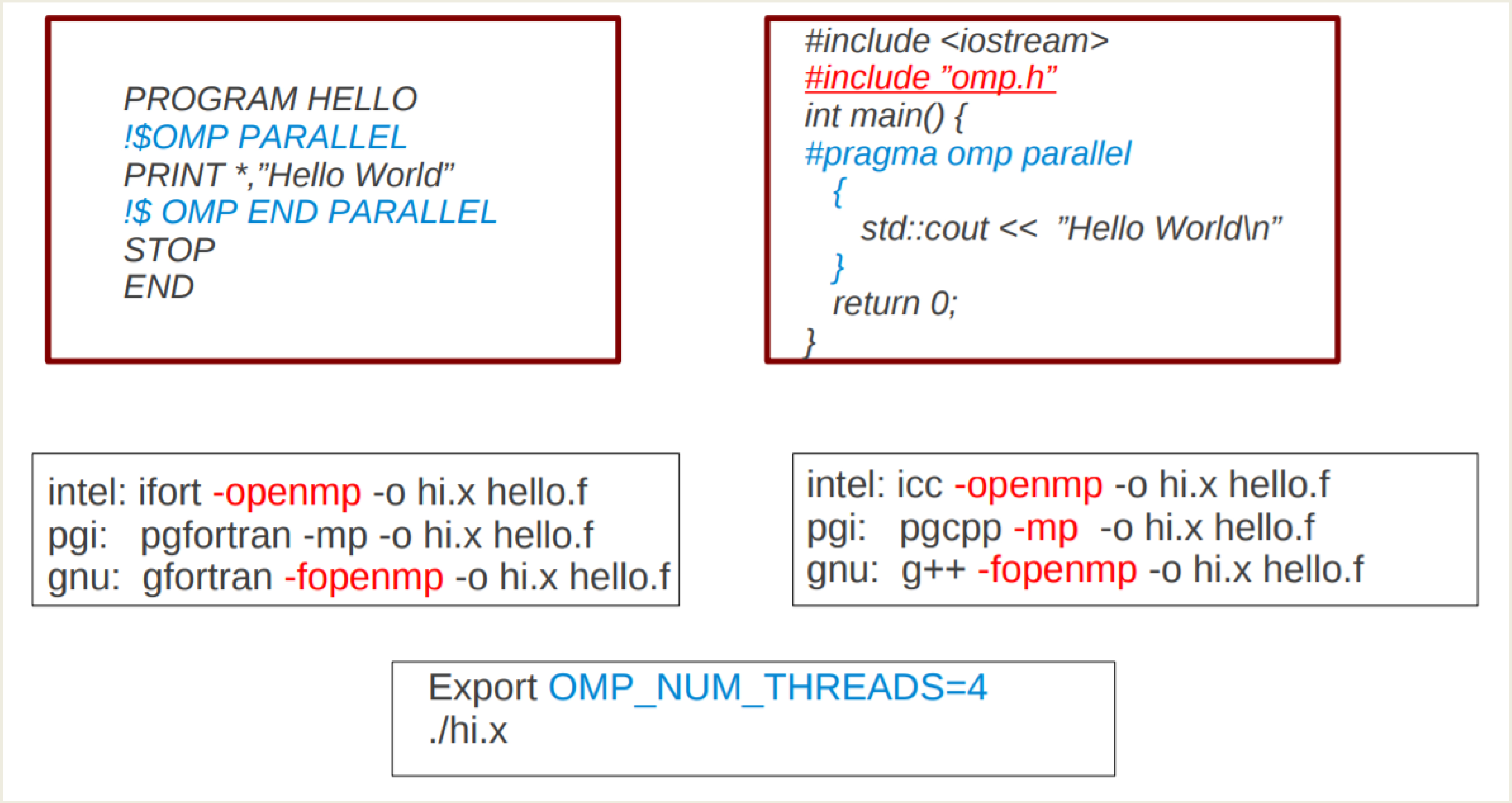


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
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 #pragma

Special preprocessor instructions.
Typically added to a system to allow behaviors that aren't part of the basic C specification.
Compilers that don't support the pragmas ignore them.

OpenMP - #pragma



```
PROGRAM HELLO
!$OMP PARALLEL
PRINT *,"Hello World"
!$ OMP END PARALLEL
STOP
END
```

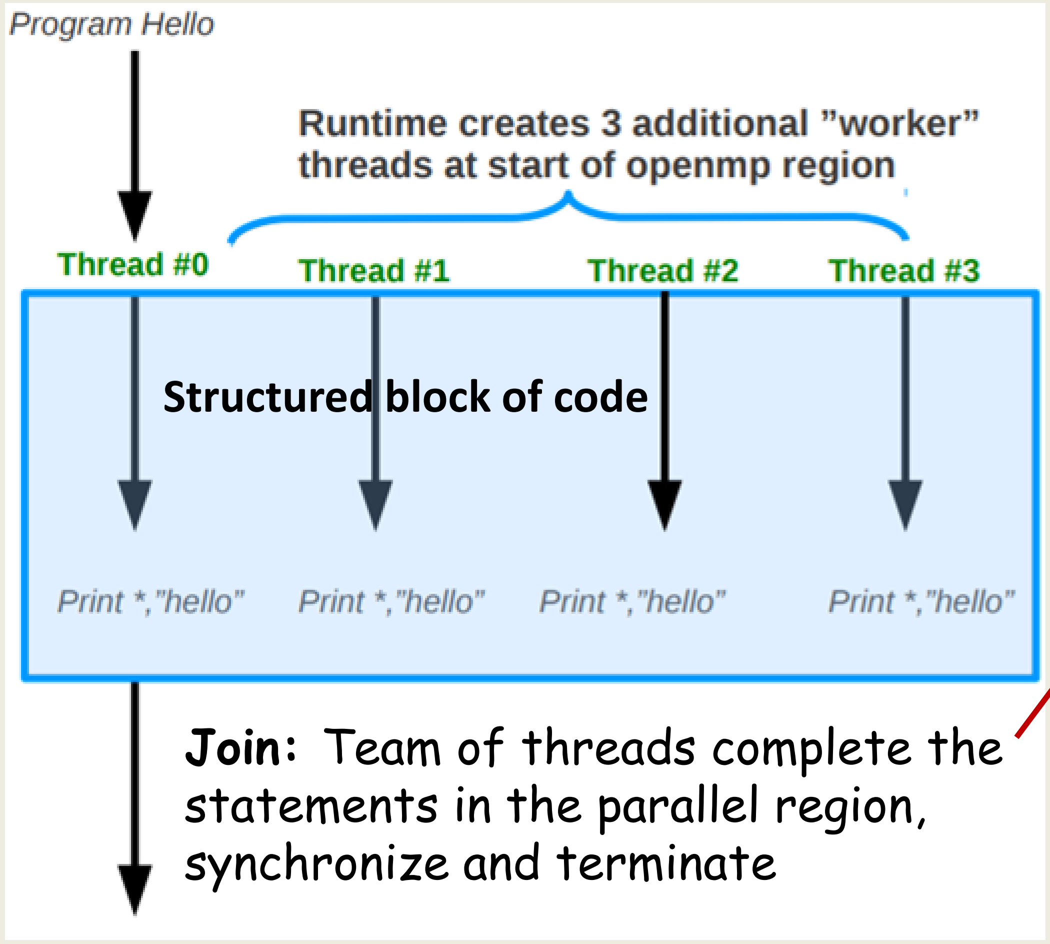
```
#include <iostream>
#include "omp.h"
int main() {
#pragma omp parallel
   {
      std::cout <<  "Hello World\n"
   }
   return 0;
}
```

```
intel: ifort -openmp -o hi.x hello.f
pgi:   pgfortran -mp -o hi.x hello.f
gnu:   gfortran -fopenmp -o hi.x hello.f
```

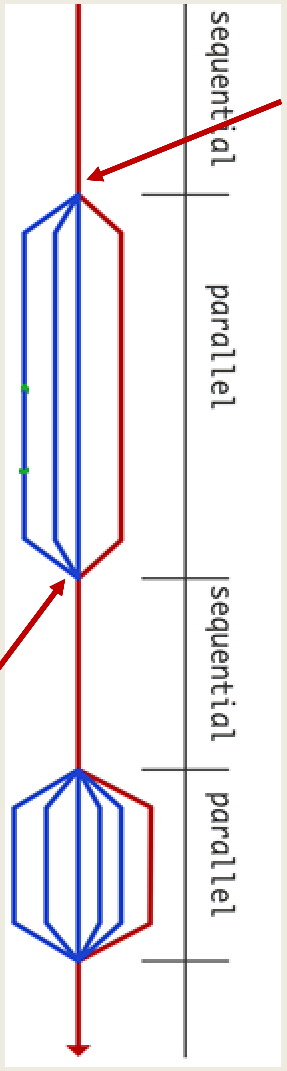
```
intel: icc -openmp -o hi.x hello.f
pgi:   pgcpp -mp  -o hi.x hello.f
gnu:   g++ -fopenmp -o hi.x hello.f
```

```
Export OMP_NUM_THREADS=4
./hi.x
```

Hello World - OpenMP



Join: Team of threads complete the statements in the parallel region, synchronize and terminate



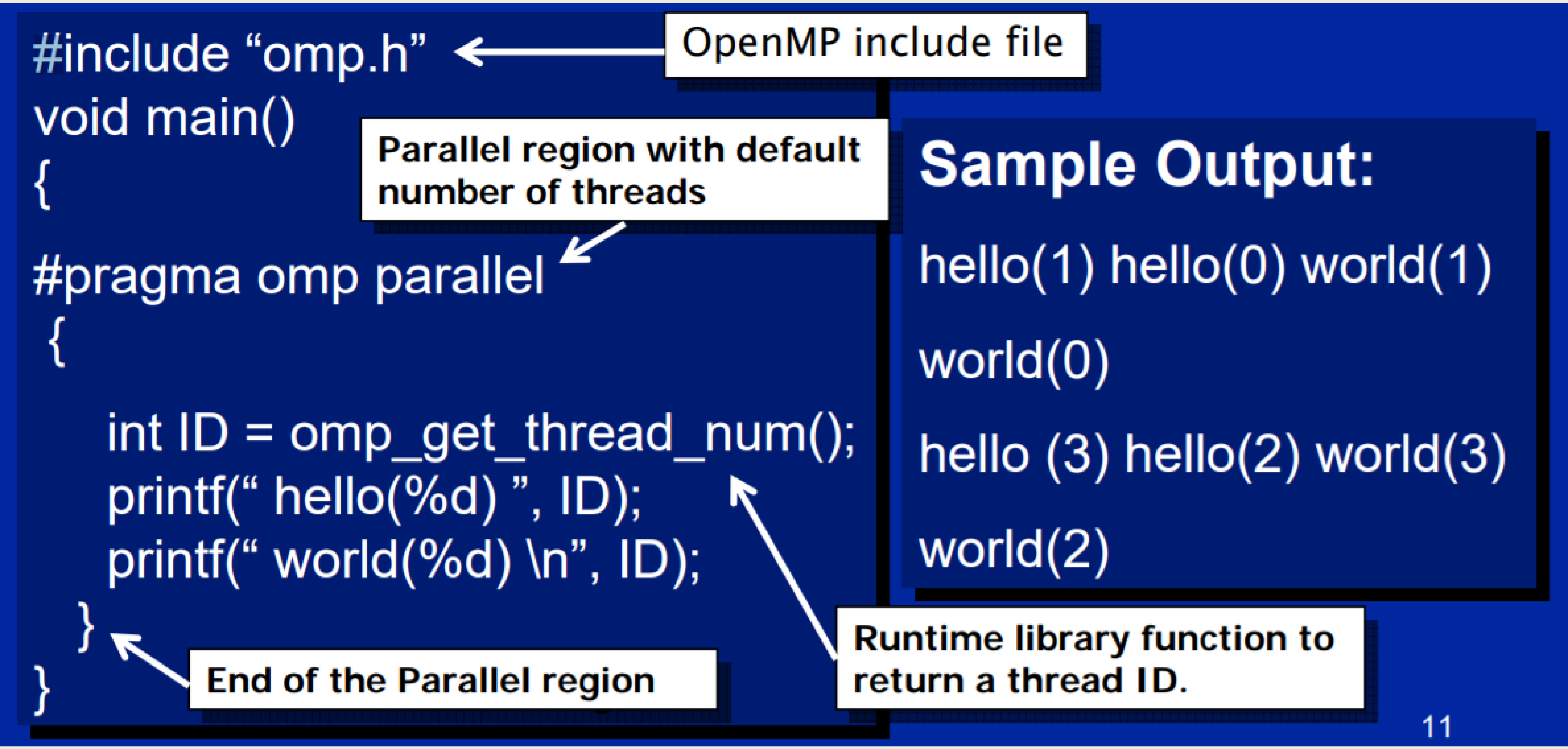
Fork: master thread creates a team of parallel ***threads***.

Threads are numbered from 0 (master thread) to N-1

Implicit ***barrier*** at the end of a parallel section.

master thread executes sequentially until the first **parallel region** is encountered.
Parallelism added incrementally until performance goals are met.

OPENMP: Basic functions



```
#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

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

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

world(0)

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

world(2)
```

11

OPENMP: basic functions

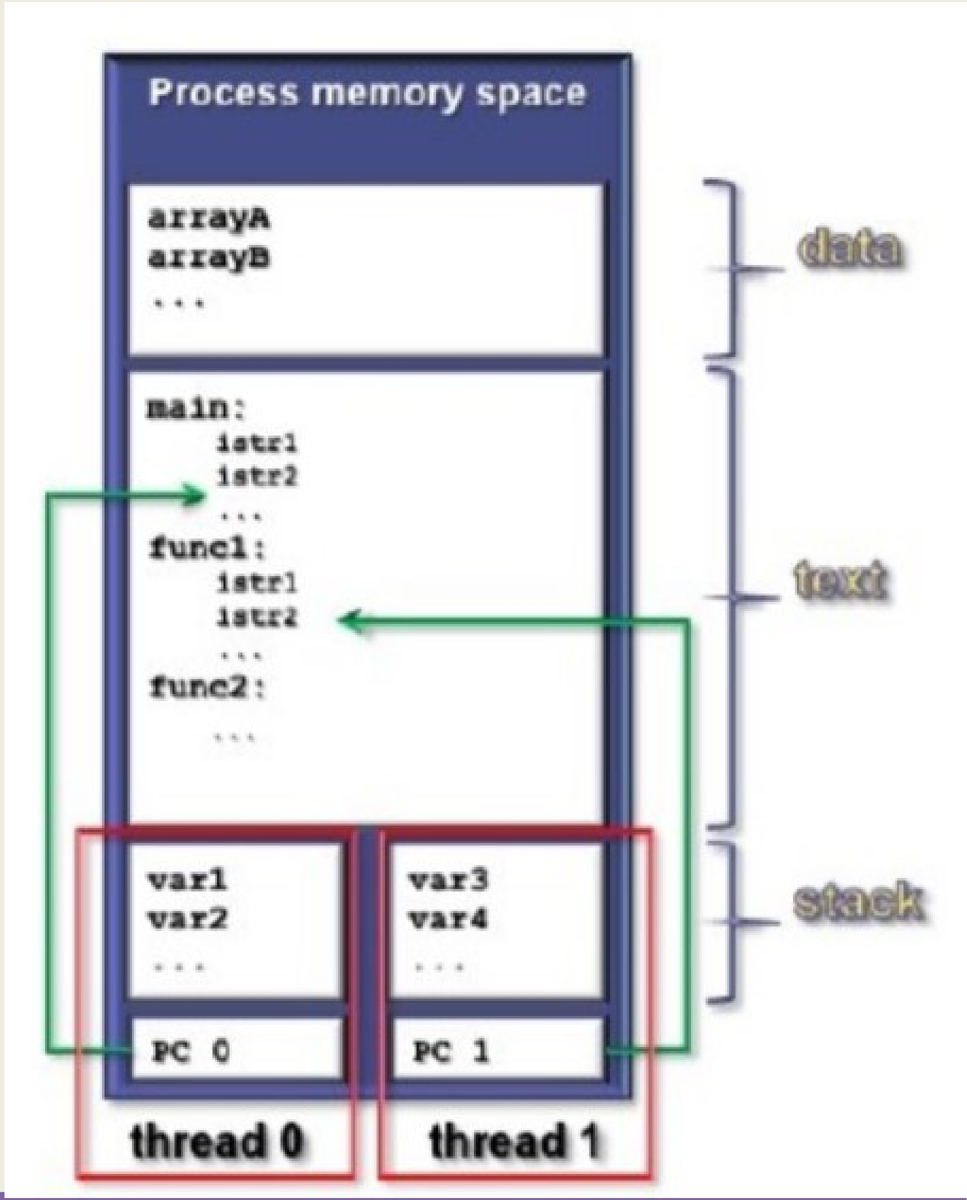
Each thread has its own stack, so it will have its own private (local) variables.

Each thread gets its own rank - **omp_get_thread_num**

The number of threads in the team - **omp_get_num_threads**

In OpenMP, **stdout** is shared among the threads, so each thread can execute the **printf** statement.

There is no scheduling of access to **stdout**, output is non-deterministic.



OPENMP: Run Time Functions

Create a 4 thread Parallel region :
Statements in the program that are enclosed by the parallel region construct are executed in parallel among the various team threads.

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

Each thread calls pooh(ID,A) for ID = 0 to 3

OpenMP Run Time Functions

Modify/check/get info about the number of threads

omp_get_num_threads() //number of threads in use

omp_get_thread_num() //tells which thread you are

omp_get_max_threads() //max threads that can be used

Are we in a parallel region?

omp_in_parallel()

How many processors in the system?

omp_get_num_procs()

Explicit locks

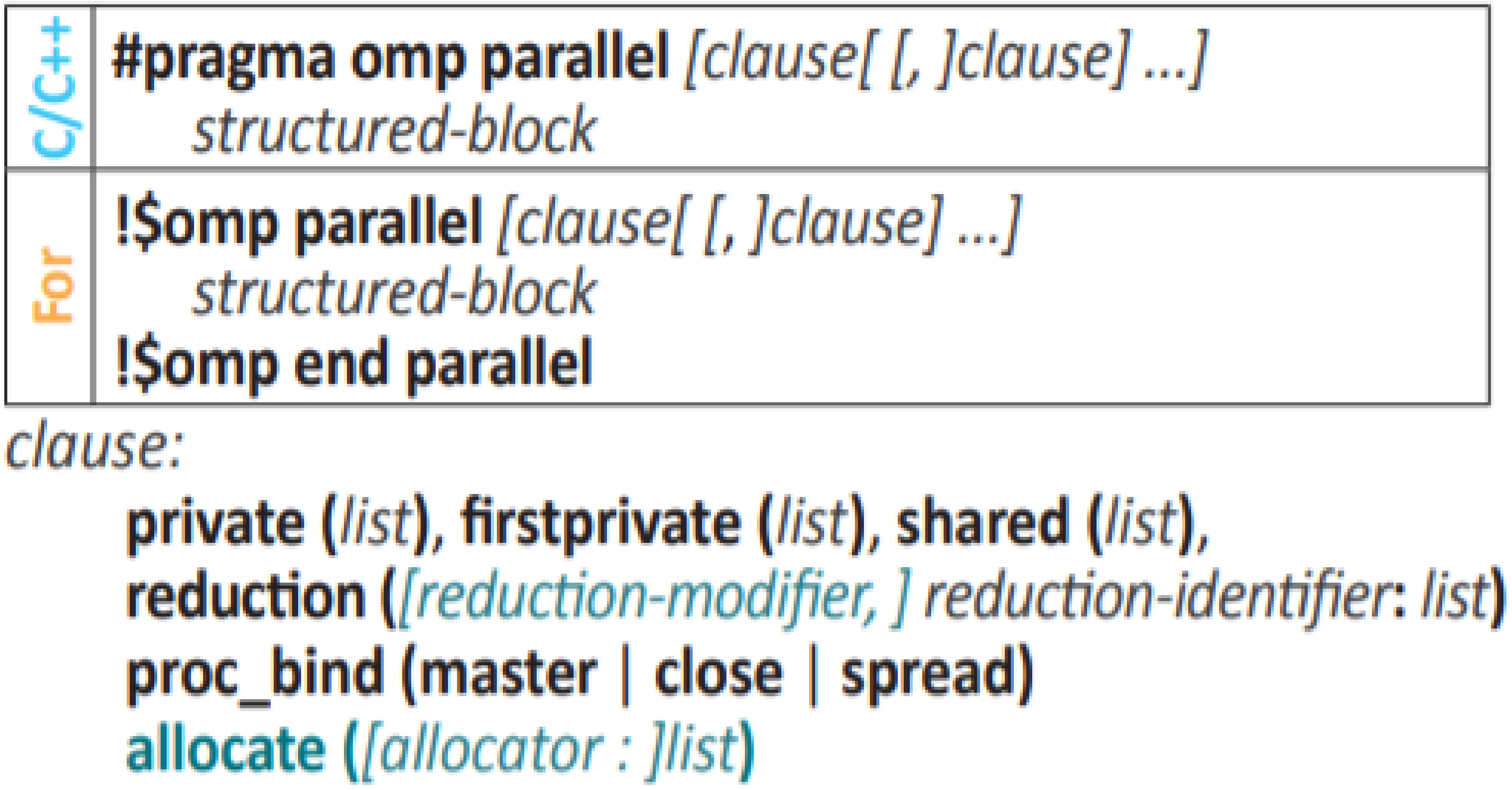
omp_[set|unset]_lock()

And several more...

OpenMP Environment Variables

Environment variable	Description
OMP_DYNAMIC	Specifies whether the OpenMP run time can adjust the number of threads in a parallel region.
OMP_NESTED	Specifies whether nested parallelism is enabled,
OMP_NUM_THREADS	Sets the maximum number of threads in the parallel region, unless overridden by omp_set_num_threads or num_threads.
OMP_SCHEDULE	Modifies the behavior of the schedule clause when schedule(runtime) is specified in a for or parallel for directive.

OpenMP parallel regions



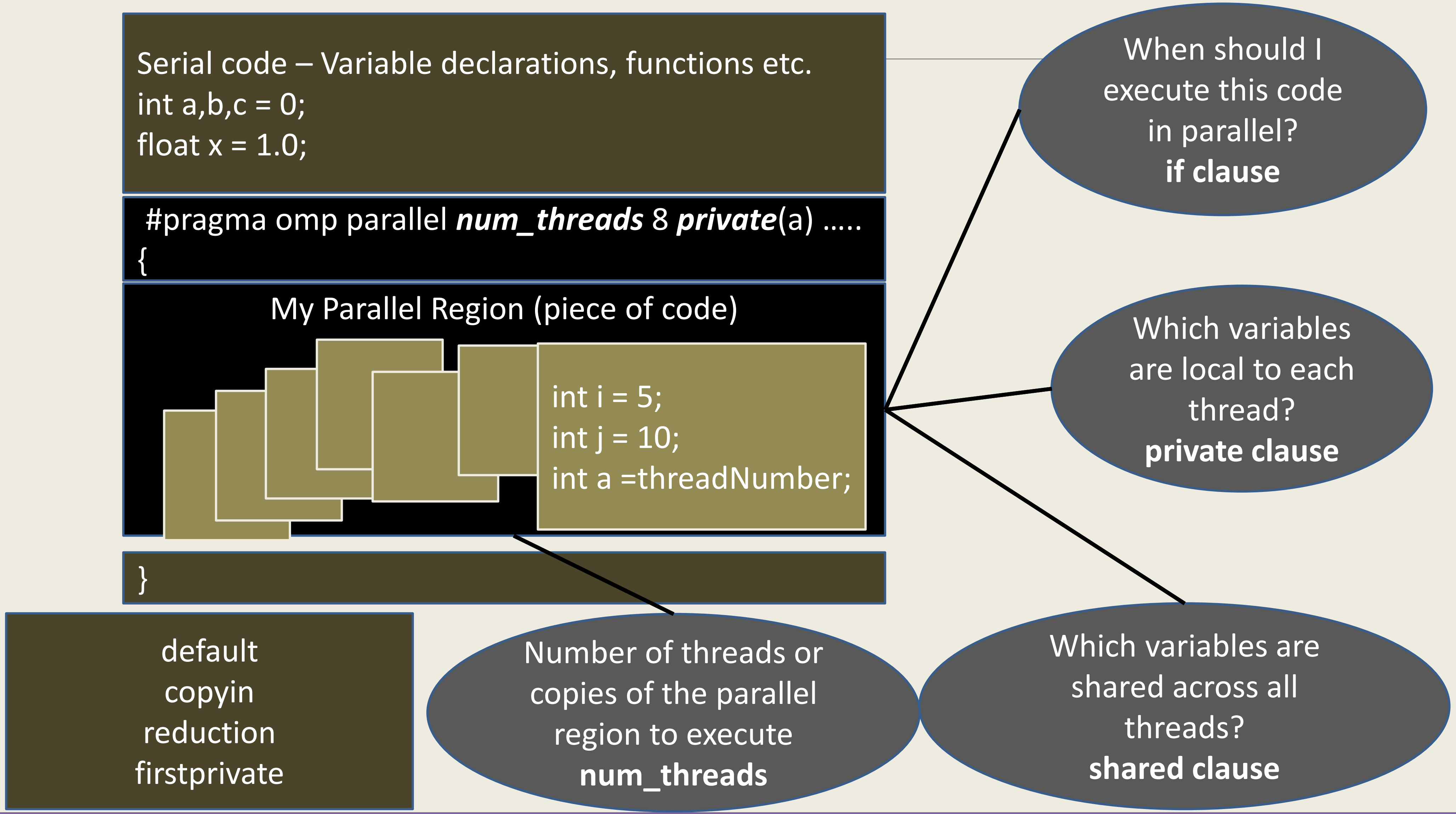
C/C++	**#pragma omp parallel** *[clause[[,]clause] ...]* *structured-block*
For	**!$omp parallel** *[clause[[,]clause] ...]* *structured-block* **!$omp end parallel**

clause:

private (*list*), **firstprivate** (*list*), **shared** (*list*),
reduction (*[reduction-modifier,] reduction-identifier*: *list*)
proc_bind (**master** | **close** | **spread**)
allocate (*[allocator :]list*)

Branching in or out of a structured block is not allowed!

OpenMP parallel regions



Serial code – Variable declarations, functions etc.

```
int a,b,c = 0;
float x = 1.0;
```

```
#pragma omp parallel num_threads 8 private(a) …..
{
```

My Parallel Region (piece of code)

```
int i = 5;
int j = 10;
int a =threadNumber;
```

```
}
```

When should I execute this code in parallel?
if clause

Which variables are local to each thread?
private clause

Number of threads or copies of the parallel region to execute
num_threads

Which variables are shared across all threads?
shared clause

default
copyin
reduction
firstprivate

OPENMP: Variable Scope

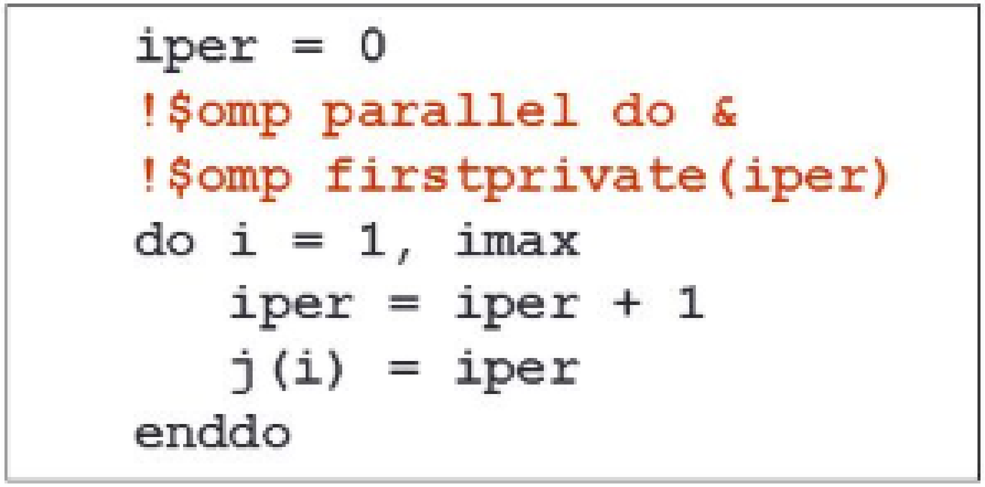
- In OpenMP, scope refers to the set of threads that can see a variable in a parallel block.
- OpenMP is a shared-memory programming model. A general rule is that any variable declared outside of a parallel region has a **shared** scope. In some sense, the "default" variable scope is shared.
- When a variable can be seen/read/written by all threads in a team, it is said to have shared scope;
- A variable that can be seen by only one thread is said to have private scope. A copy of the private variable in each thread.
- Loop variables in an **omp for** are **private**
- Local variables in the parallel region are private
- Change default behavior by using the clause **default(shared) or default(private)**

OpenMP: private Clause

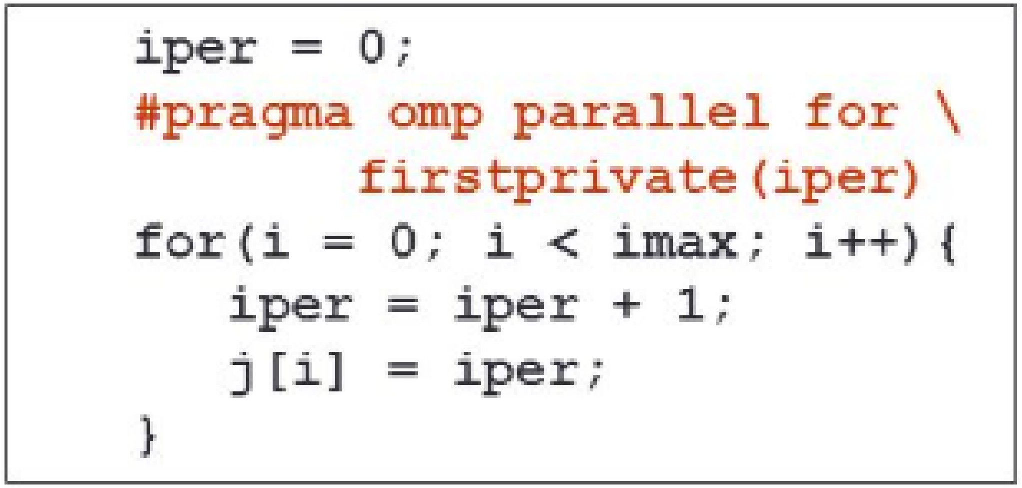
```
void* work(float* c, int N) {
    float x, y; int i;
#pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
}
```

- Reproduce the private variable for each thread.
- Variables are not initialized.
- The value that Thread1 stores in x is different from the value Thread2 stores in x

OpenMP: firstprivate Clause



```
iper = 0
!$omp parallel do &
!$omp firstprivate(iper)
do i = 1, imax
    iper = iper + 1
    j(i) = iper
enddo
```

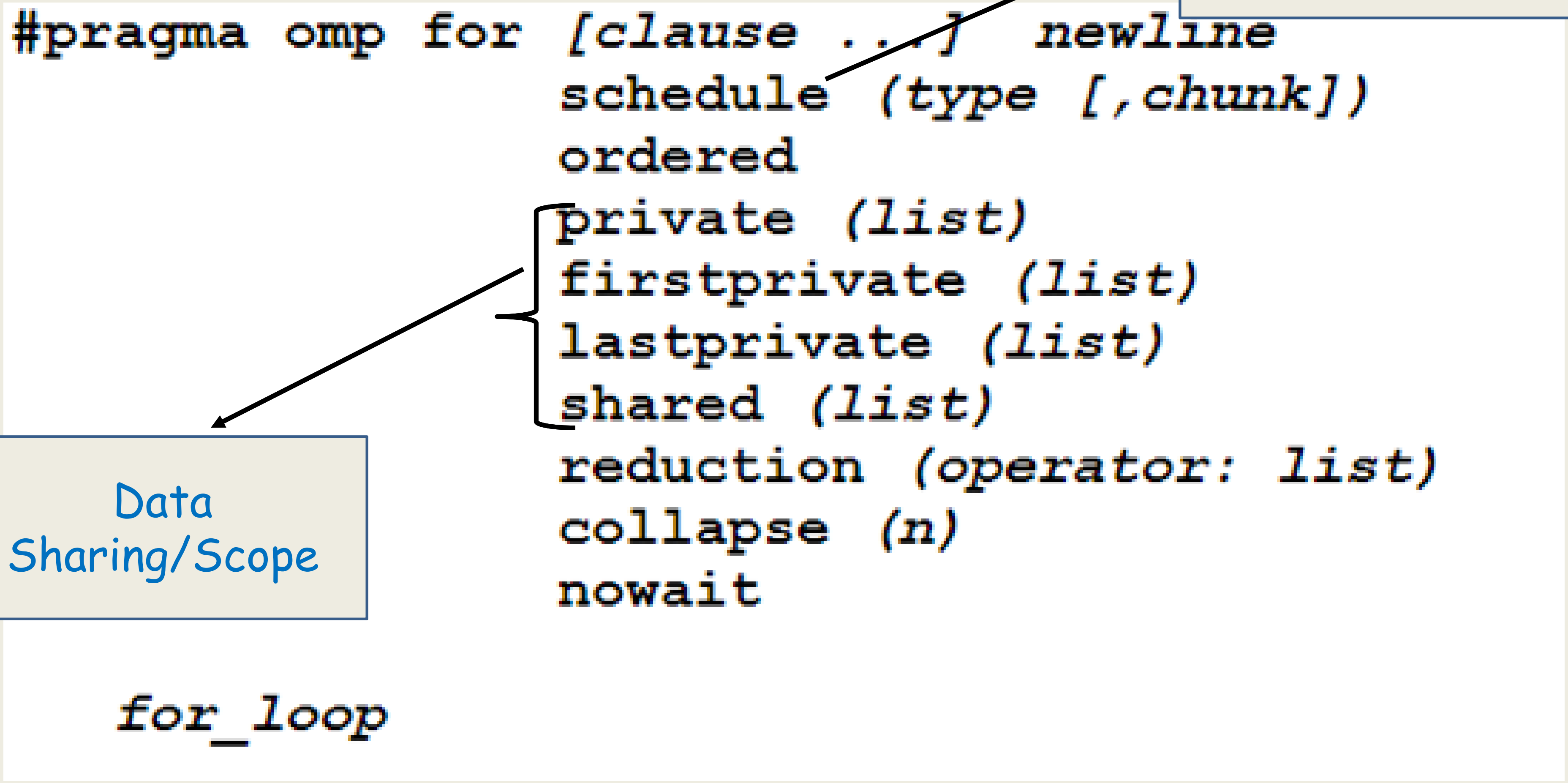


```
iper = 0;
#pragma omp parallel for \
        firstprivate(iper)
for(i = 0; i < imax; i++){
    iper = iper + 1;
    j[i] = iper;
}
```

- Creates private memory location for **iper** for each thread.
- Copy value from master thread to each memory location
- While initial value is same, it can be changed by threads and subsequently Thread 0 Thread 1 and 2.. Might have different values of the firstprivate variable

OpenMP: Clauses & Data Scoping

Schedule Clause



```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

Data Sharing/Scope

Matrix Vector Multiplication

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

$$\begin{array}{|c|c|c|c|}\hline a_{00} & a_{01} & \cdots & a_{0,n-1} \\ \hline a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline \end{array} \begin{array}{|c|}\hline x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{n-1} \\ \hline \end{array} = \begin{array}{|c|}\hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline \end{array}$$

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

```
#pragma omp parallel
for (i=0; i < m; i++)
{    y[i] =0.0;
     for (j=0; j < SIZE; j++)
     y[i] += (A[i][j] * x[j]);
}
```

Will this work?

Matrix Vector Multiplication

$$\begin{array}{|c|c|c|c|}\hline a_{00} & a_{01} & \cdots & a_{0,n-1} \\ \hline a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline \end{array} \begin{array}{|c|}\hline x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{n-1} \\ \hline \end{array} = \begin{array}{|c|}\hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline \end{array}$$

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

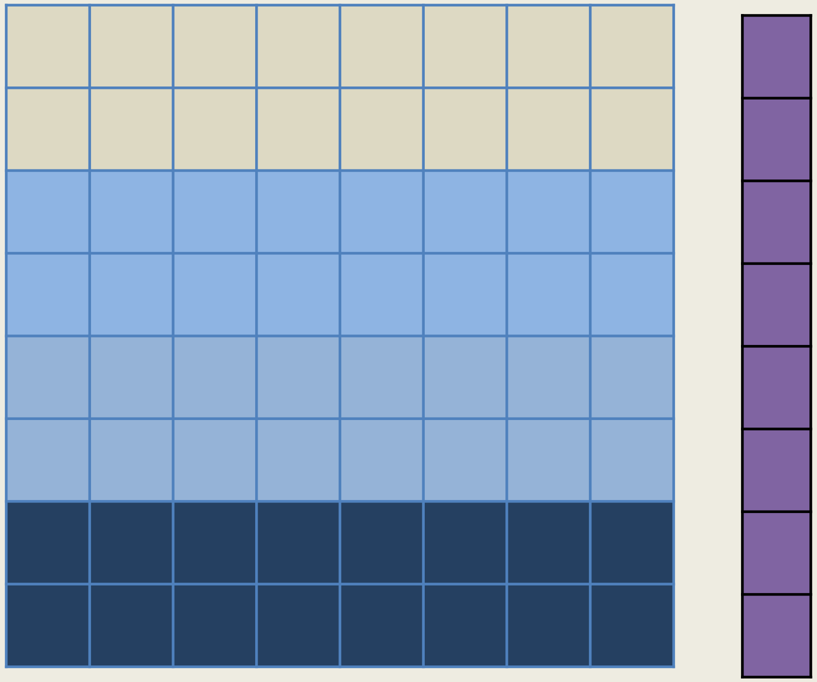
```
#pragma omp parallel
for (i=0; i < m; i++)
{   y[i] =0.0;
    for (j=0; j < SIZE; j++)
    y[i] += (A[i][j] * x[j]);
}
```

Independent tasks:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

Will this work?

Matrix Vector Multiplication



Matrix Rows = N (= 8)
Number of Threads = T (=4)
Number of Rows processed by thread = N/T
Thread 0 => rows 0,1,2,3,…(N/T – 1)
Thread 1 => rows N/T, N/T+1…… 2*N/T - 1
Thread t => rows t, t+1, t+2, …. (t*N/T -1)

```
#pragma omp parallel shared(A,x,y,SIZE) \
private(tid,i,j,istart,iend)
{
    tid = omp_get_thread_num();
    nid = omp_get_num_threads();
    istart = tid*SIZE/nid;
    iend = (tid+1)*SIZE/nid;

    for (i=istart; i < iend; i++)
    {
        for (j=0; j < SIZE; j++)
         y[i] += (A[i][j] * x[j]);

      printf(" thread %d did row %d\t
      y[%d]=%.2f\t",tid,i,i,y[i]);
     }
} /* end of parallel construct */
```

Matrix Vector Multiplication

```
omp_set_num_threads(4)
#pragma omp parallel shared(A,x,y,SIZE)
{
    #pragma omp for
    for (int i=0; i < SIZE; i++)
    {
          for (int j=0; j < SIZE; j++)
           y[i] += (A[i][j] * x[j]);
    }
 } /* end of parallel construct */
```

```
Matrix Rows = N (= 8)
Number of Threads = T (=4)
Number of Rows processed by thread = N/T
Thread 0 => rows 0,1,2,3,…(N/T – 1)
Thread 1 => rows N/T, N/T+1……  2*N/T  - 1
Thread t => rows t, t+1, t+2, …. (t*N/T -1)
```

#pragma omp for must be inside a parallel region (#pragma omp parallel)

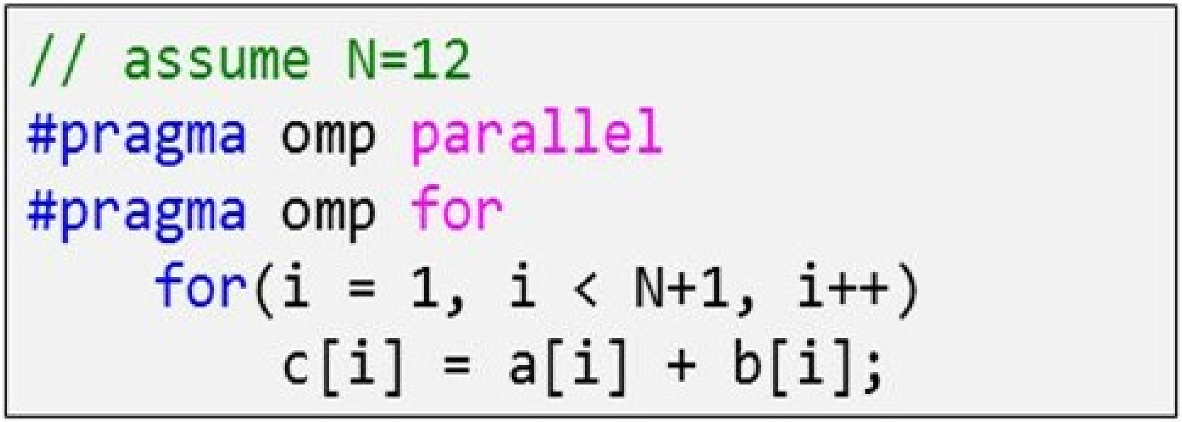
No new threads are created but the threads already created in the enclosing parallel region are used.

The system automatically parallelizes the for loop by dividing the iterations of the loop among the threads.

User can control how to divide the loop iterations among threads by using the schedule clause.

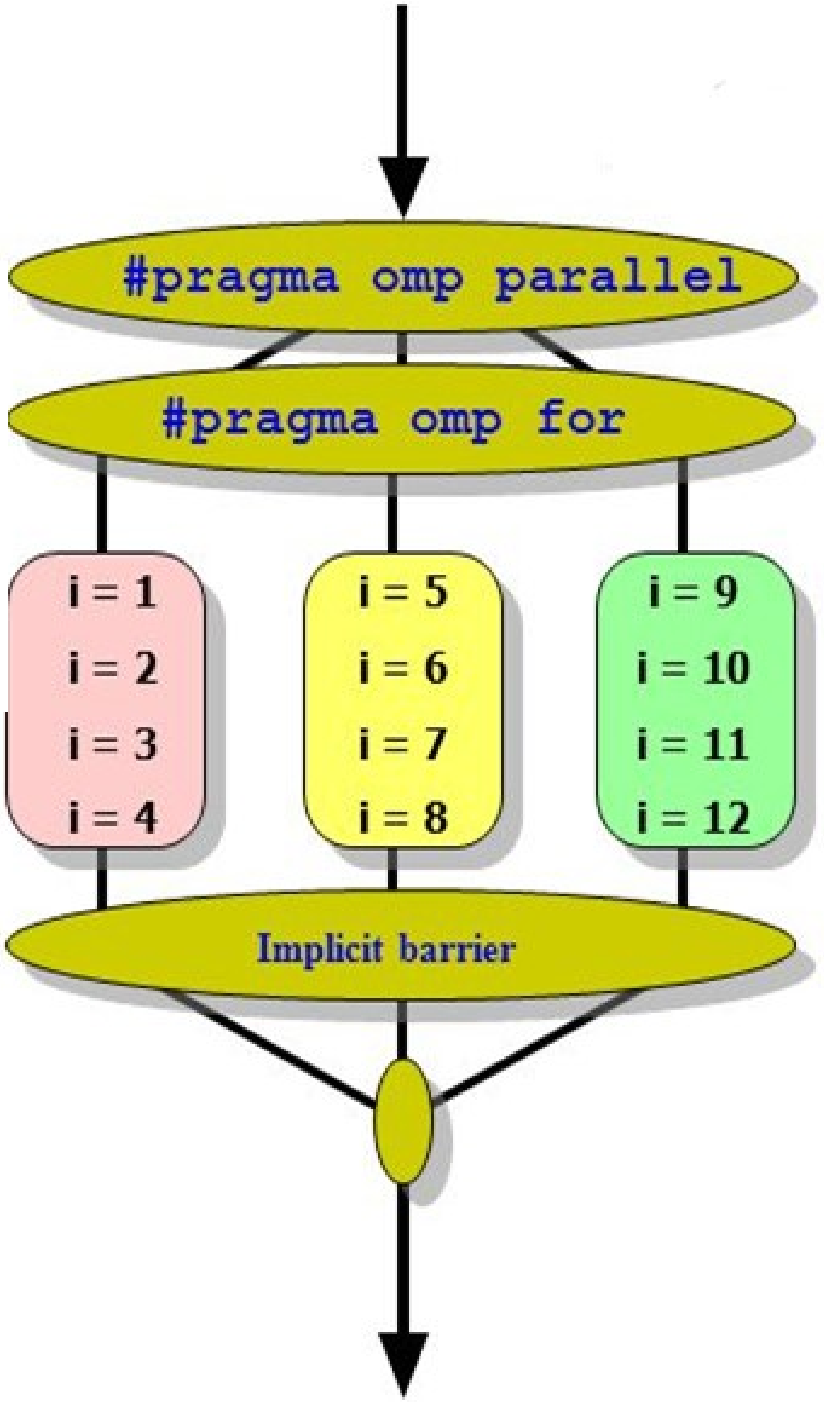
User controlled Variable Scope

#pragma omp for
#pragma omp parallel for



```
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1, i < N+1, i++)
        c[i] = a[i] + b[i];
```

- OpenMP takes care of partitioning the iteration space for you.
- Threads are assigned independent sets of iterations.
- There is no implied barrier upon entry to a work-sharing construct, There is an implied barrier at the end of a work sharing construct



OpenMP: Work Sharing

Data parallelism

Large amount of data elements and each data element (or possibly a subset of elements) needs to be processed to produce a result. When this processing can be done in parallel, we have data parallelism (for loops)

Task parallelism

A collection of tasks that need to be completed. If these tasks can be performed in parallel you are faced with a task parallel job

Work Sharing: omp for

- Sequential code to add two vectors

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

- **OpenMP implementation 1 (not desired)**

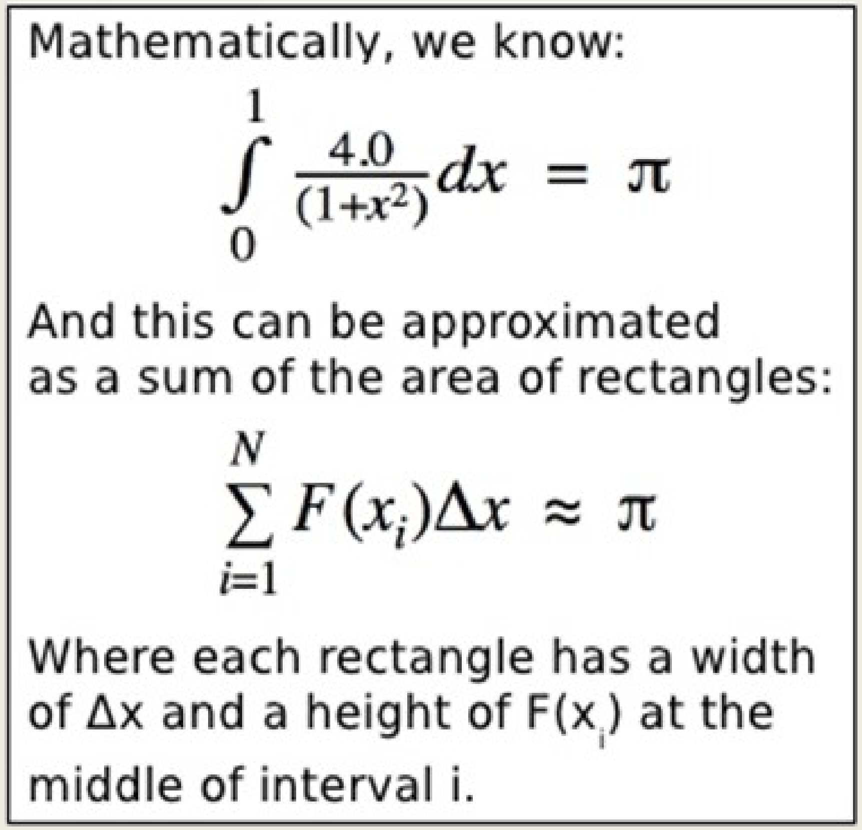
```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
     istart = id*N/Nthrds;
     iend = (id+1)*N/Nthrds;
     if(id == Nthrds-1) iend = N;
     for(I = istart; i<iend; i++) {c[i] = b[i]+a[i];}
}
```

- **A worksharing for construct to add vectors**

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

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

Computing ∏ by method of Numerical Integration



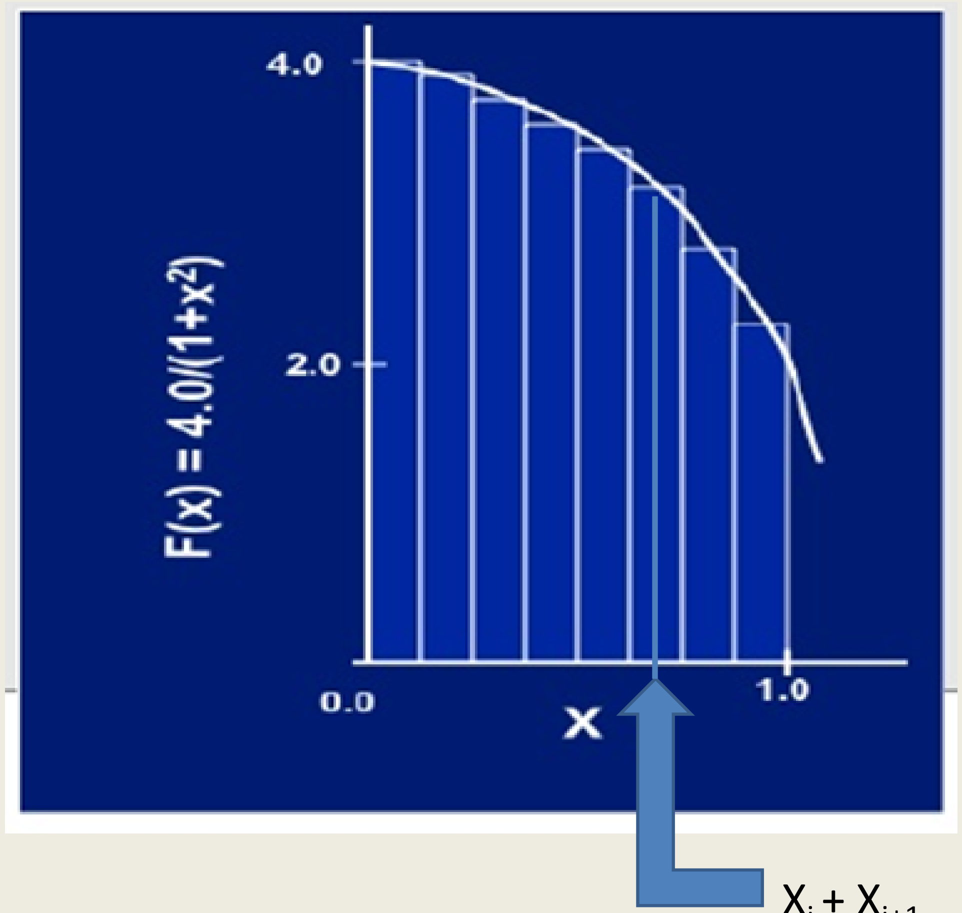
Mathematically, we know:

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

And this can be approximated as a sum of the area of rectangles:

$$\sum_{i=1}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has a width of Δx and a height of $F(x_i)$ at the middle of interval i.



$$\frac{X_i + X_{i+1}}{2}$$

Divide the interval (x axis) [0,1] into N parts.
Area of each rectangle is x * y [x = 1/N, y = 4/ $(1+x^2)$] =[1/N] *4/ $(1+x^2)$
Approximation of x as midpoint of the interval before computing Y

Serial Code

```
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 = 0; I <= num_steps; i++)
      {
            x = (I + 0.5) * step;
            sum = sum +  4.0 / (1.0 + x*x);
     }
     pi = step * sum
}
```

task1

task2

1. Computation of the areas of individual rectangles
2. Adding the areas of rectangles.

There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 2

Computing Π by method of Numerical Integration

```
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 = 0; I <= num_steps; i++) {
           x = (I + 0.5) * step;
          sum = sum + 4.0 / (1.0 + x*x);
    }
    pi = step * sum
}
```

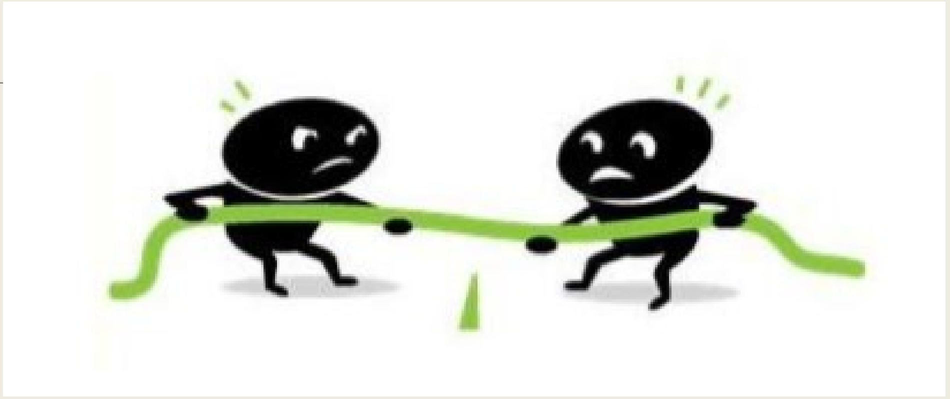
Serial Code

```
#include  <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000;
double step;
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 for shared(sum)
private(x)
     for (I = 0; I <= num_steps; i++) {
          x = (I + 0.5) * step;
          sum = sum + 4.0 / (1.0 + x*x);
    }
    pi = step * sum
}
```

Parallel Code

Race Condition

```
#pragma omp parallel for
shared(global_result) private(x, myresult)
    for (I = 0; I <= num_steps; i++) {
        x = (I + 0.5) * step;
        myresult = 4.0 / (1.0 + x*x);
        global_result += myresult;
    }
```



Time	Thread 0	Thread 1
0	`global_result = 0` to register	finish `my_result`
1	`my_result = 1` to register	`global_result = 0` to register
2	add `my_result` to `global_result`	`my_result = 2` to register
3	store `global_result = 1`	add `my_result` to `global_result`
4		store `global_result = 2`

Unpredictable results when two (or more) threads attempt to simultaneously execute: **global_result += myresult**

Handling Race Conditions

```
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for shared(sum)
private(x)
     for (I = 0; I <= num_steps; i++) {
          x = (I + 0.5) * step;
          #pragma omp critical
          sum = sum + 4.0 / (1.0 + x*x);
     }
```

Mutual Exclusion:
Only one thread at a time executes the statement
sum = sum + 4.0 / (1.0 + x*x);

Use **synchronization** to protect data conflicts.
- Mutual Exclusion (#pragma omp critical)
- Mutual Exclusion (#pragma omp atomic)

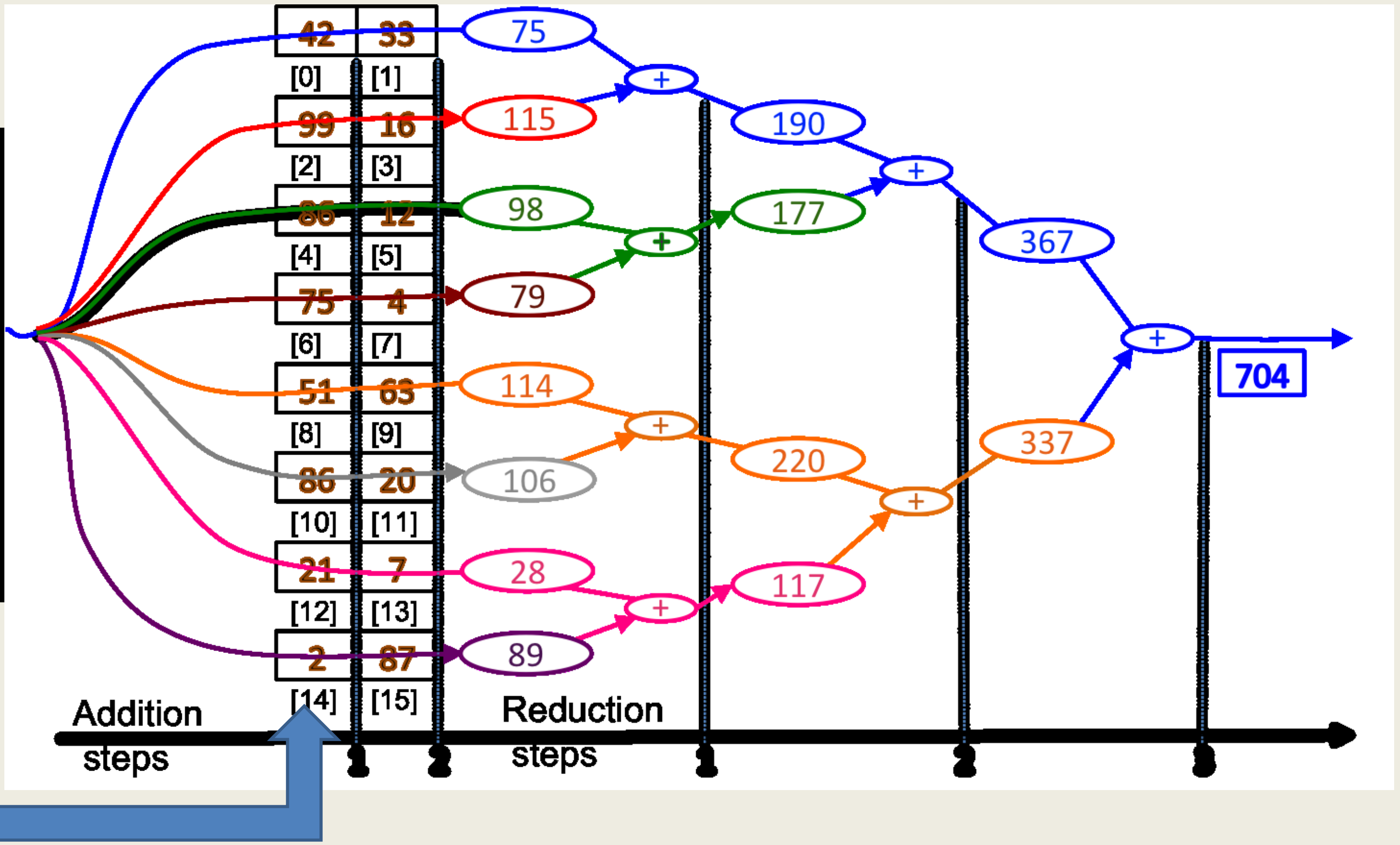
Synchronization could be expensive so:
Change how data is accessed to minimize the need for synchronization.

OpenMP: Reduction

```
sum = 0;
Set_omp_num_threads(8)
#pragma omp parallel for
reduction (+:sum)
for (int i = 0; i < 16; i++)
{
    sum += a[i]
}
```

Thread0 => iteration 0 & 1
Thread1 => iteration 2 & 3
………

Thread local/private



One or more variables that are private to each thread are subject of reduction operation at the end of the parallel region.

#pragma omp for reduction(operator : var)

Operator: + , * , - , & , | , && , ||, ^

Combines multiple local copies of the **var** from threads into a single copy at master.

Computing Π by method of Numerical Integration

```
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 = 0; I <= num_steps; i++) {
             x = (I + 0.5) * step;
            sum = sum + 4.0 / (1.0 + x*x);
     }
     pi = step * sum
}
```

Serial Code

```
#include  <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000;
double step;
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 for reduction(+:sum)
private(x)
      for (I = 0; I <= num_steps; i++) {
             x = (I + 0.5) * step;
             sum += 4.0 / (1.0 + x*x);
     }
     pi = step * sum
}
```

Parallel Code

omp for Parallelization

Can all loops be parallelized?
Loop iterations have to be independent.

Simple Test: If the results differ when the code is executed backwards, the loop cannot by parallelized!

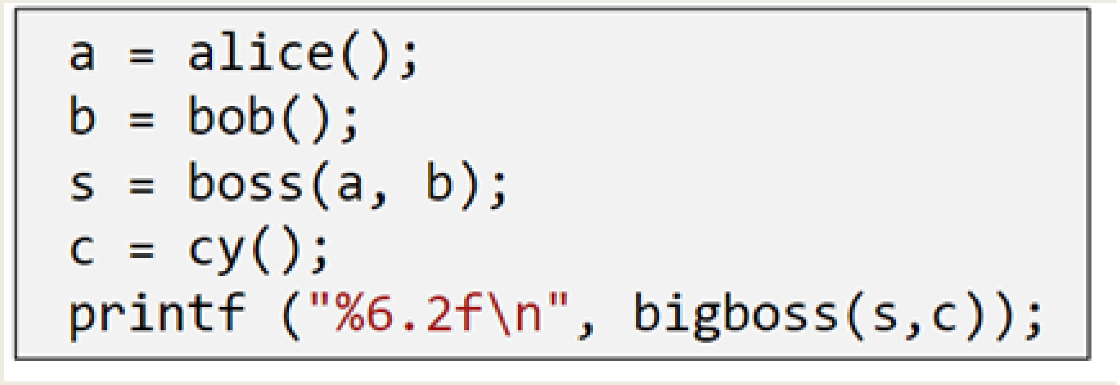
```
for (int i = 2; i < 10;  i++)
{
      x[i] = a * x[i-1] + b
}
```

Between 2 Synchronization points, if atleast 1 thread writes to a memory location, that atleast 1 other thread reads from => The result is non-deterministic

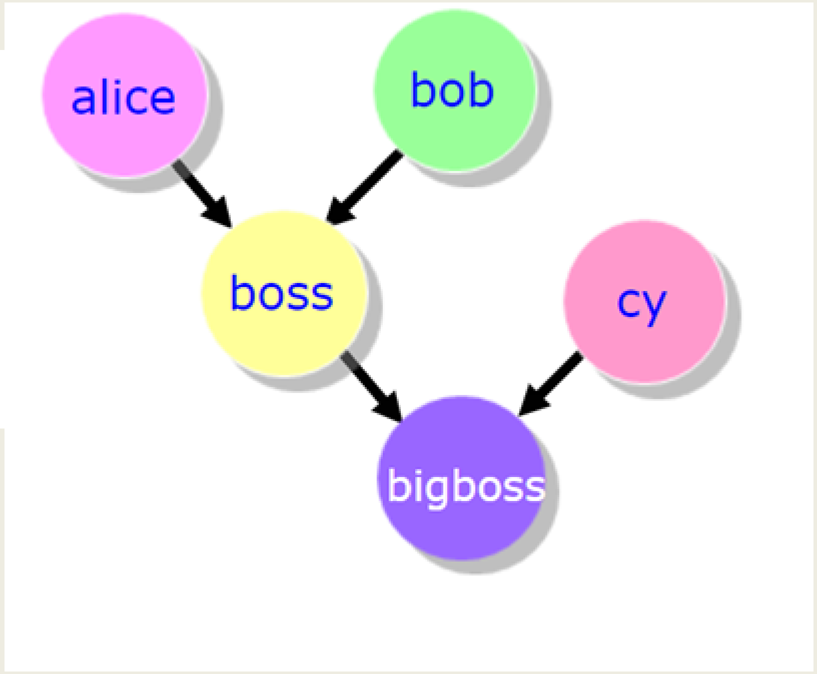
Work Sharing: sections

SECTIONS directive is a non-iterative work-sharing construct.

It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
Each SECTION is executed ONCE by a thread in the team.



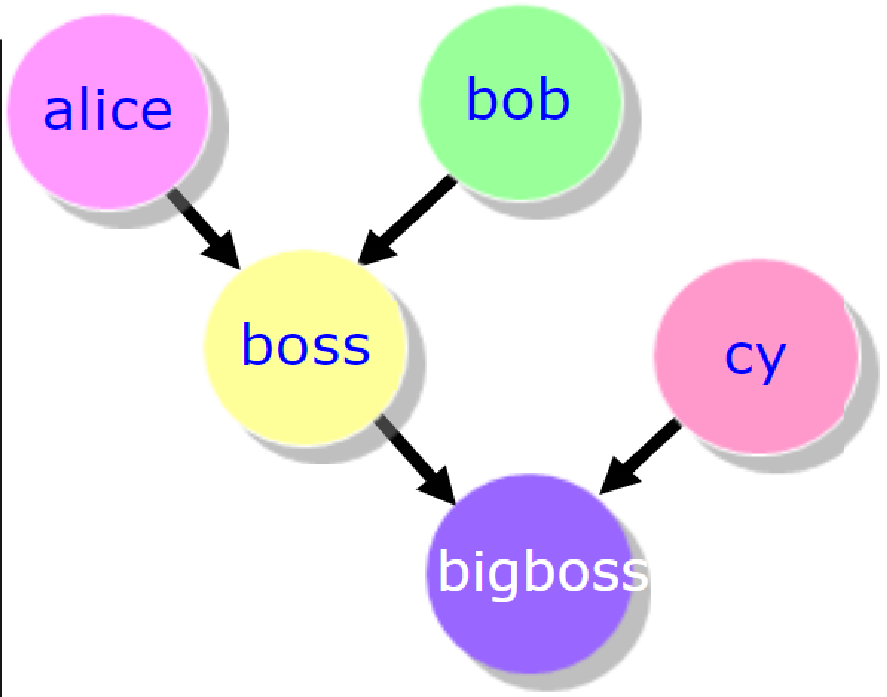
```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n", bigboss(s,c));
```



Work Sharing: sections

```
#pragma omp parallel sections
{
#pragma omp section
    double a = alice();
#pragma omp section
    double b = bob();
#pragma omp section
    double c = cy();
}

 double s = boss(a, b);
 printf ("%6.2f\n", bigboss(s,c));
```



OpenMP: lastprivate Clause

```
!$OMP DO PRIVATE(I)
LASTPRIVATE(B)
DO i = 1, 1000
B = i
ENDDO
!$OMP END DO
!—value of B here is
1000
```

```
!$OMP SECTIONS
LASTPRIVATE(B)
!$OMP SECTION
      B = 2
!$OMP SECTION
      B = 4
!$OMP SECTION
      D = 6
!$OMP END SECTIONS
```

- Creates private memory location for each thread.
- Does not initialize the private variable.
- The sequentially last iteration of the associated loops, or the lexically last section construct [...] to the original list item.

Work Sharing: tasks

#pragma omp task [clauses]……

- Tasks allow to parallelize irregular problems (Unbounded loops & Recursive algorithms)
- A task has - Code to execute – Data environment (It owns its data) – Internal control variables – An assigned thread that executes the code and the data
- Each encountering thread packages a new instance of a task (code and data)
- Some thread in the team executes the task at some later time

Work Sharing: tasks

Fibonacci series:
f(1) = 1
f(2) = 1
f(n) = f(n-1) + f(n-2)

```
/* serial code to compute Fibonacci */
int fib(int n)
{
   int i, j;
   if(n < 2) return n;
   i = fib(n-1);
   j = fib(n-2);
   return (i+j);
}
int main(){
   int n = 8;
   printf("fib(%d) = %d\n", n, fib(n);
}
```

```
Static int fib(int n){
   int i, j, id;
   if(n < 2)
      return n;
   #pragma omp task shared (i) private (id)
   {
      i = fib(n-1);
   }
   #pragma omp task shared (j) private (id)
   {
      j = fib(n-2);
   }
   return (i+j);
}
```

#pragma omp taskwait

Work Sharing: single

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.

- May be useful when dealing with sections of code that are not thread safe (such as I/O)

```
!$OMP SINGLE [clause ...]
    PRIVATE (list)
    FIRSTPRIVATE (list)
   block
!$OMP END SINGLE [ NOWAIT ]
```

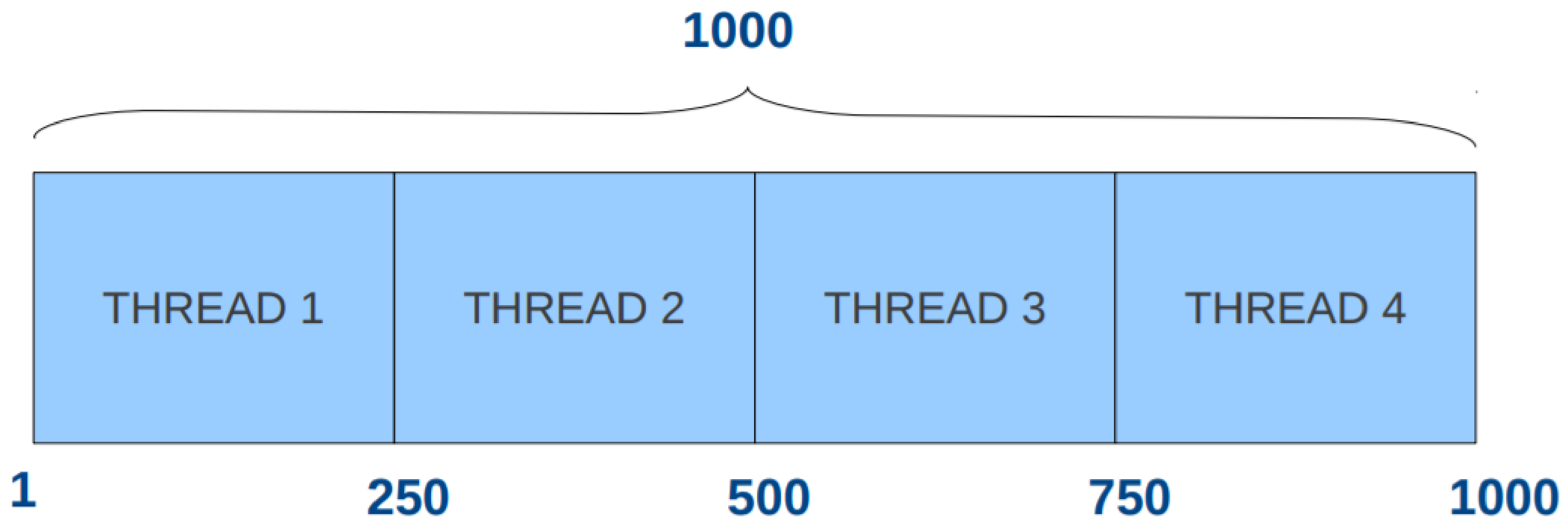
```
#pragma omp single [clause ...] newline
private (list) firstprivate (list) nowait
structured_block
```

Schedule Clause

How is the work is divided among threads?
Directives for work distribution

Although the OpenMP standard does not specify how a loop should be partitioned most compilers split the loop in N/p (N #iterations, p #threads) chunks by default. This is called a static schedule (with chunk size N/p)

For example, suppose we have a loop with 1000 iterations and 4 omp threads. The loop is partitioned as follows:



Schedule Clause: Types

A schedule kind is passed to an OpenMP loop schedule clause:

- provides a hint for how iterations of the corresponding OpenMP loop should be assigned to threads in the team of the OpenMP region surrounding the loop.
- Five kinds of schedules for OpenMP loop1:

  static
  dynamic
  guided
  auto
  runtime

- The OpenMP implementation and/or runtime defines how to assign chunks to threads of a team given the kind of schedule specified by as a hint.

Schedule Clause

STATIC: Iterations of a loop are divided into chunks of size **ceiling**(*iterations/threads*). Each thread is assigned a separate chunk.
STATIC, N: Iterations of a loop are divided into chunks of size **N**. Each chunk is assigned to a thread in *round-robin* fashion. ***N*** >= 1 (integer expression)

DYNAMIC: Iterations of a loop are divided into chunks of size 1.
Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.
DYNAMIC, N: Same as above, all chunks are set to size **N**

GUIDED: Chunks are made progressively smaller until a chunk size of one is reached. The first chunk is of size **ceiling**(*iterations/threads*). Remaining chunks are of size **ceiling**(*iterations_remaining/threads*).Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.
GUIDED, N: Minimum chunk size is N

AUTO: Delegated the decision of the scheduling to the compiler and/or runtime system
RUNTIME: Scheduling policy is determined at run time. OMP_SCHEDULE/ OMP_SET_SCHEDULE

OpenMP: Synchronization

- The programmer needs finer control over how variables are shared.
- The programmer must ensure that threads do not interfere with each other so that the output does not depend on how the individual threads are scheduled.
- In particular, the programmer must manage threads so that they read the correct values of a variable and that multiple threads do not try to write to a variable at the same time.

- MASTER, CRITICAL, BARRIER, FLUSH, TASKWAIT, ORDERED, NOWAIT

Synchronization Constructs

To impose order constraints and protect shared data.

Achieved by **Barriers** & **Mutual Exclusion**

1) Barriers (Task Dependencies)

Implicit : Sync points exist at the end of

- parallel –necessary barrier – cant be removed
- for – can be removed by using the **nowait** clause
- sections – can be removed by using the **nowait** clause
- single – can be removed by using the **nowait** clause

Explicit : Must be used when ordering is required

```
#pragma omp barrier
```

each thread waits until all threads arrive at the barrier

Synchronization: Barrier

```
#pragma omp parallel private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier

    #pragma omp for
    for(i=0;i<N;i++)
    {
        C[i]=big_calc3(i,A);
    }

    #pragma omp for nowait
    for(i=0;i<N;i++)
    {
        B[i]=big_calc2(C, i);
    }
    A[id] = big_calc4(id);
}
```

Explicit Barrier

Implicit Barrier at end of parallel region

No Barrier
nowait cancels barrier creation

Data Dependencies

OpenMP assumes that there is NO data-dependency across jobs running in parallel

When the **omp parallel** directive is placed around a code block, it is the programmer's responsibility to make sure data dependency is ruled out

Synchronization Constructs

2) Mutual Exclusion (Data Dependencies)

- **Critical Sections** : Protect access to shared & modifiable data, allowing ONLY ONE thread to enter it at a given time
  - #pragma omp critical
  - #pragma omp atomic – special case of **critical**, less overhead
- **Locks**

Only one thread updates this at a time

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
#pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
#pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

OPENMP Synchronization: review

PRAGMA	DESCRIPTION
#pragma omp taskwait !$OMP TASKWAIT	Specifies a wait on the completion of child tasks generated since the beginning of the current task
#pragma omp critical !$OMP CRITICAL !$OMP END CRITICAL	Code within the block or pragma is only executed on one thread at a time.
#pragma omp critical !$OMP ATOMIC !$OMP END ATOMIC	Provides a mini-CRITICAL section. specific memory location must be updated atomically (Atomic statements)
#pragma omp barrier !$OMP BARRIER !$OMP END BARRIER	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.

OPENMP Synchronization: review

PRAGMA	DESCRIPTION
#pragma omp for **ordered** *[clauses...] (loop region)* #pragma omp **ordered** *structured_block*	Used within a DO / for loop Iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor. Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.
#pragma omp flush *(list)*	Synchronization point at which all threads have the same view of memory for all shared objects. FLUSH is implied for barrier parallel - upon entry and exit critical - upon entry and exit ordered - upon entry and exit for - upon exit sections - upon exit single - upon exi

Running OpenMP code

Controlling the number of threads at runtime

- The default number of threads = number of **online** processors on the machine.
- C shell : **setenv OMP_NUM_THREADS number**
- Bash shell: **export OMP_NUM_THREADS = number**
- Runtime OpenMP function omp_set_num_threads(4)
- Clause in #pragma for parallel region

Execution Timing

```
#include omp.h
stime = omp_get_wtime();
longfunction();
etime = omp_get_wtime();
total = etime-stime;
```