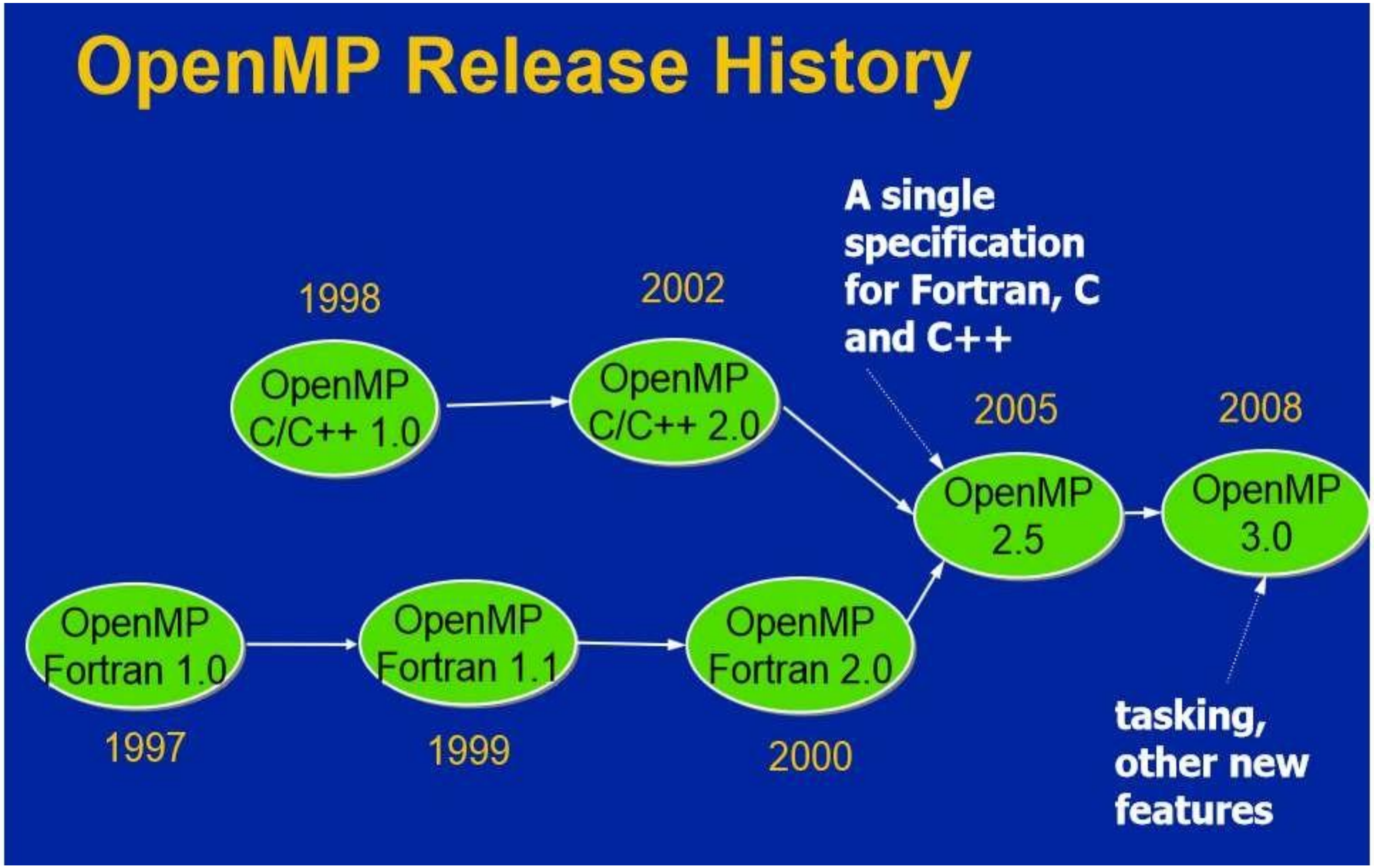
OPENMP

Open Specifications for Multi Processing

What is OpenMP?

- De-facto standard API for writing shared memory parallel applications in C, C++, and Fortran
- Consists of:
  - Compiler Directives
  - Runtime Routines
  - Environment variables
- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)
- Version 4.0 has been released July 2013


OpenMP Release History
A single specification for Fortran, C and C++
1998
OpenMP C/C++ 1.0
2002
OpenMP C/C++ 2.0
2005
OpenMP 2.5
2008
OpenMP 3.0
OpenMP Fortran 1.0
1997
OpenMP Fortran 1.1
1999
OpenMP Fortran 2.0
2000
tasking, other new features

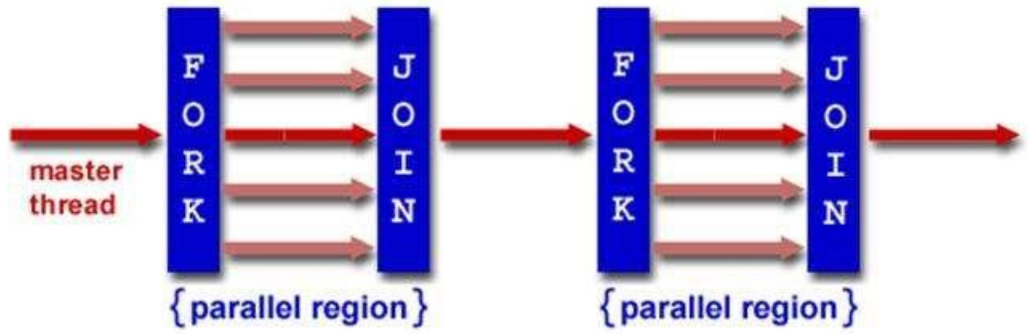
When to consider

- When compiler cannot find parallelism
- The granularity is not high enough

- USE EXPLICIT PARALLELIZATION - OpenMP

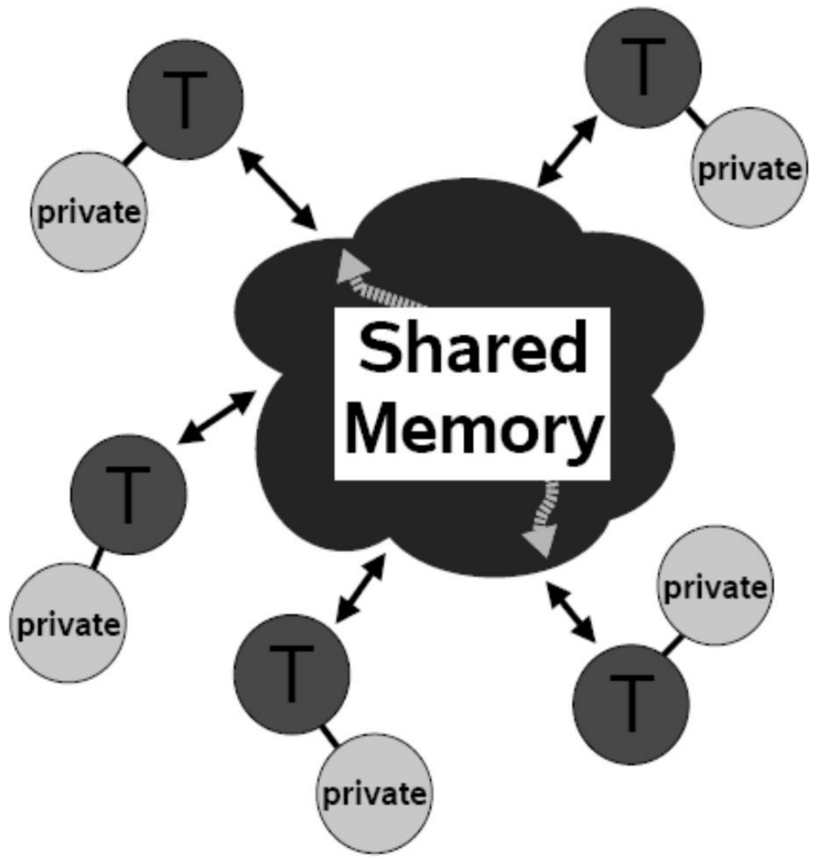
Memory Model

- **Shared Memory, Thread Based Parallelism**
- **Explicit Parallelism**
- **Fork - Join Model**



- **Compiler Directive Based**
- **Nested Parallelism Support**
- **Dynamic Thread**
- **Memory Model : Flush often**

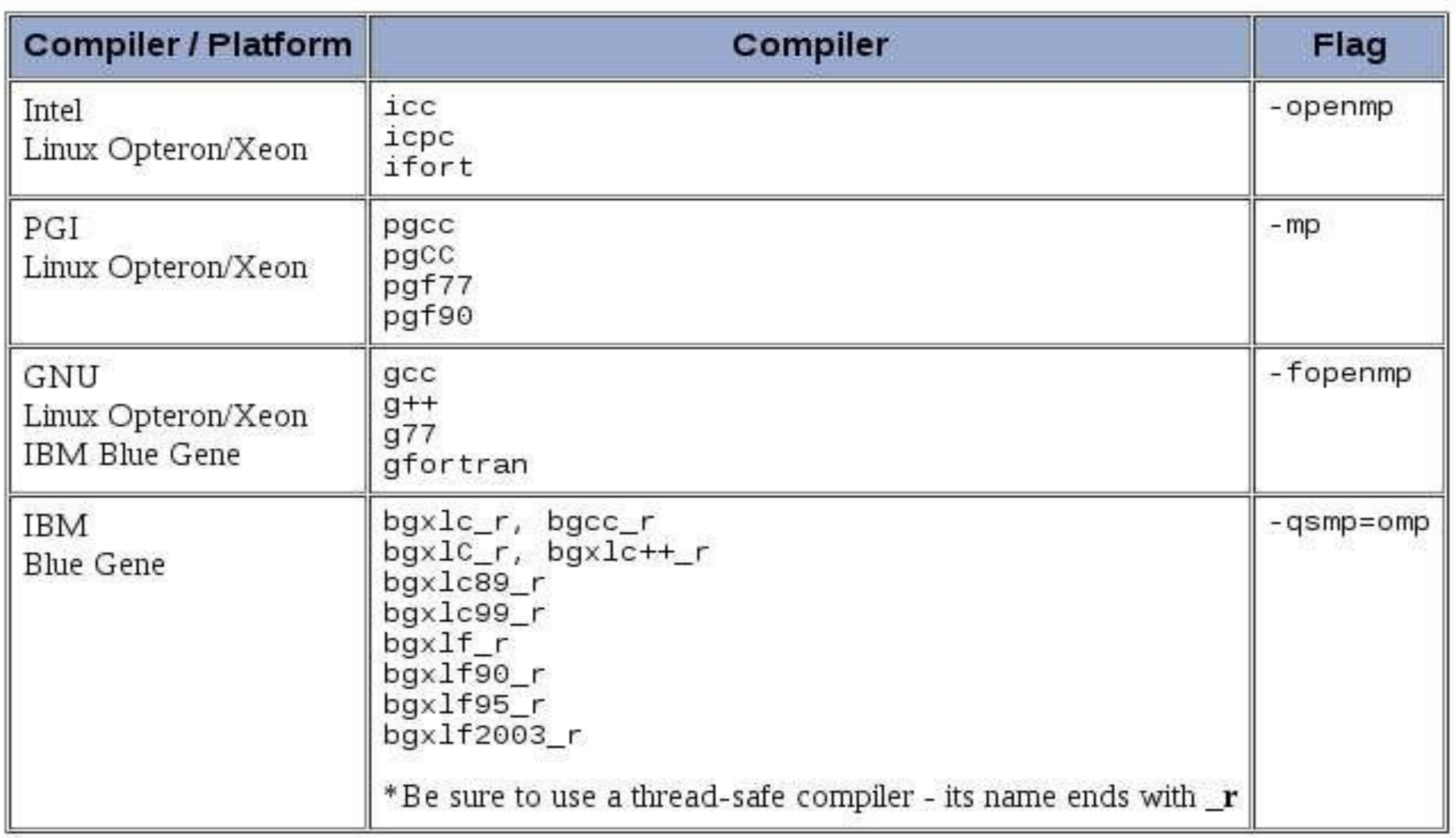
Contd..



- Data is private or shared.
- All threads have access to same globally shared memory.
- Shared data accessible by all threads.
- Private accessed only by owned threads.
- Data transfer is transparent to programmer.
- Synchronization takes place, but it is almost implicit.

❖ Sample program

Compilation



Compiler / Platform	Compiler	Flag

Intel
Linux Opteron/Xeon
	`icc`
	`icpc`
	`ifort`
		`-openmp`

PGI
Linux Opteron/Xeon
	`pgcc`
	`pgCC`
	`pgf77`
	`pgf90`
		`-mp`

GNU
Linux Opteron/Xeon
IBM Blue Gene
	`gcc`
	`g++`
	`g77`
	`gfortran`
		`-fopenmp`

IBM
Blue Gene
	`bgxlc_r, bgcc_r`
	`bgxlC_r, bgxlc++_r`
	`bgxlc89_r`
	`bgxlc99_r`
	`bgxlf_r`
	`bgxlf90_r`
	`bgxlf95_r`
	`bgxlf2003_r`
	
	*Be sure to use a thread-safe compiler - its name ends with **_r**
		`-qsmp=omp`

- ❖ **GNU Compiler Example :**
  - gcc -o omp_helloc -fopenmp omp_hello.c
- ❖ **IBM AIX compiler :**
  - xlc – omp_helloc -qsmp=omp omp_hello.c
- ❖ **Portland group compiler:**
  - pgcc -o omp_helloc -mp omp_hello.c
- ❖ **Intel Compiler Example:**
  - icc -o omp_helloc -openmp omp_hello.c

Advantages of OpenMP

- **Good performance and scalability**
  - ✓**If you do it right ....**
- **De-facto and mature standard**
- **An OpenMP program is portable**
  - ✓**Supported by a large number of compilers**
- **Requires little programming effort**
- **Allows the program to be parallelized incrementally**

When can it be parallelized

- **Scenario**
  - **On one processor**
  - **On two processor**

  - **Their order of execution must not matter!**

  - **Example 1**

```
a=1;
b=2;
```

Example 2

```
a=2;
b=a;
```

Components of OpenMP 2

Compiler Directives

- Parallel Construct
- Work Sharing
- Synchronization
- Data Environment
  - ✓private
  - ✓first private
  - ✓last private
  - ✓shared
  - ✓reduction

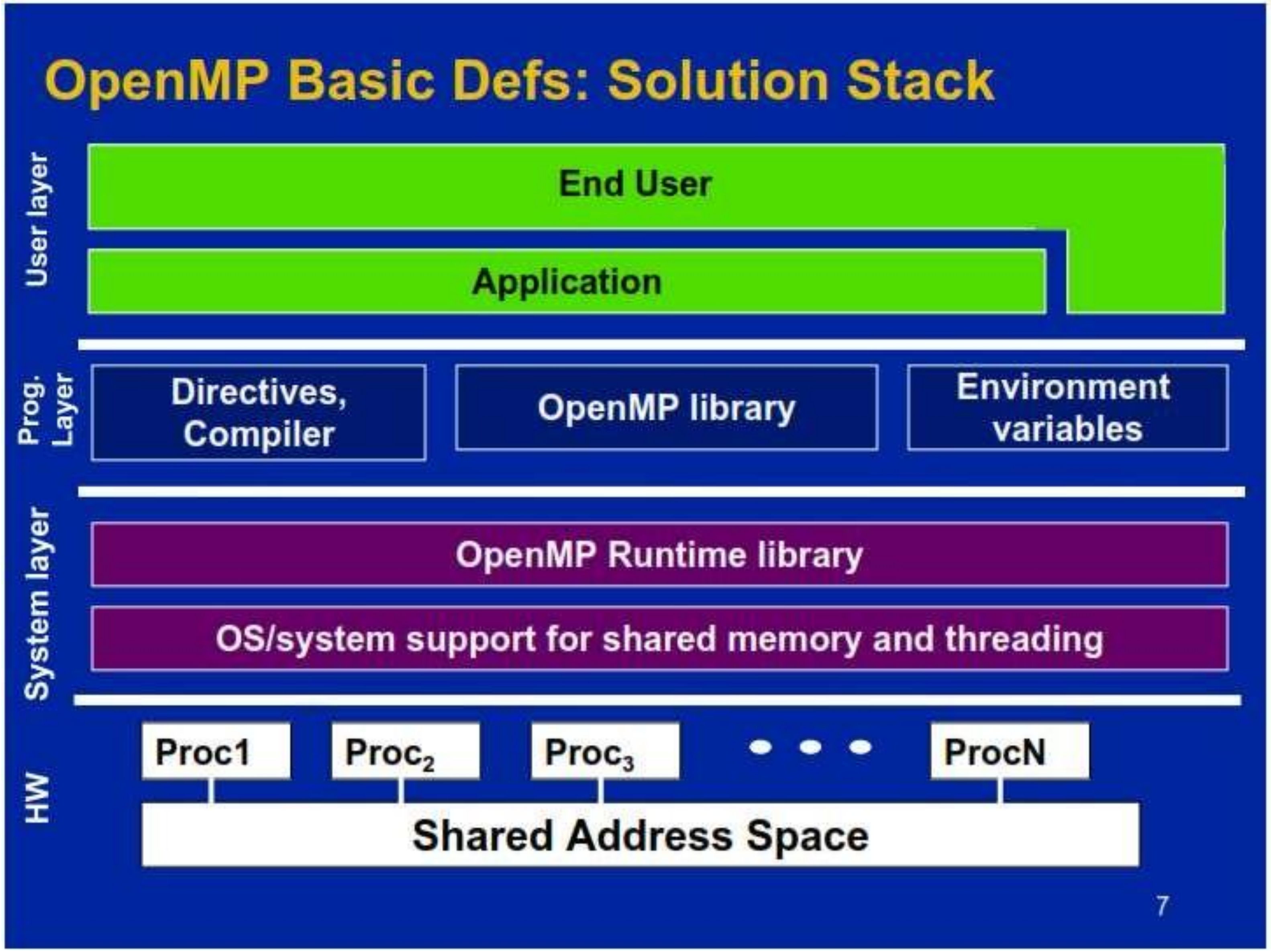
Environment Variables

- Number of threads
- Scheduling Type
- Nested parallelism
- Dynamic Thread
- Adjustment

Runtime Library routines

- Number of threads
- Thread ID
- Dynamic thread
- adjustment
- Nested parallelism

OpenMP Solution Stack



OpenMP Directives

- **#pragma omp directive-name [clause, clause..] new-line**
  - **Eg: #pragma omp parallel default(shared) private(beta,pi)**

- **General Rules:**
  - **Case sensitive**
  - **Compiler Directives follow C/C++ standards**
  - **Only one directive-name to be specified per directive**
  - **Each directive applies to at most one succeeding statement.**
  - **Use ("\") for continuing on succeeding lines.**

Directives

I. PARALLEL Region Construct

- A parallel region is a block of code that will be executed by multiple threads

```
#pragma omp parallel   [clause ...] newline
                            if (scalar_expression)
                            private (list)
                            shared (list)
                            firstprivate (list)
                            reduction (operator: list)
                           default (shared | none)
                           copyin (list)
                           num_threads (integer-expression)
 structured_block
```

Parallel Directive…

- Main thread creates a team of threads and becomes the master of the team.
- The master is a member of that team and has thread id 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section.
- If any thread terminates within a parallel region, all threads in the team terminate.

❖ Restrictions:

- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted

Parallel Directive...

❖ **How Many Threads?**

- **The number of threads in a parallel region is determined by the following factors, in order of precedence:**
  - ✓ **Evaluation of the IF clause**
  - ✓ **Setting of the NUM_THREADS clause**
  - ✓ **Use of the omp_set_num_threads() library function**
  - ✓ **Setting of the OMP_NUM_THREADS environment variable**
  - ✓ **Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).**
- **Threads are numbered from 0 (master thread) to N-1**

```
#include <omp.h>

void subdomain(float *x, int istart, int ipoints)
{
  int i;

  for (i = 0; i < ipoints; i++)
      x[istart+i] = 123.456;
}

void sub(float *x, int npoints)
{
    int iam, nt, ipoints, istart;

#pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt =  omp_get_num_threads();
        ipoints = npoints / nt;    /* size of partition */
        istart = iam * ipoints;   /* starting array index */
        if (iam == nt-1)      /* last thread may do more */
          ipoints = npoints - istart;
        subdomain(x, istart, ipoints);
    }
}

int main()
{
    float array[10000];

    sub(array, 10000);

    return 0;
}
```

Directives contd..

❖ **Data Scoping Attribute Clauses**

The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include:

- ✓ **PRIVATE**
- ✓ **FIRSTPRIVATE**
- ✓ **LASTPRIVATE**
- ✓ **SHARED**
- ✓ **DEFAULT**
- ✓ **REDUCTION**
- ✓ **COPYIN**

Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables

Attribute Scoping

❖ **private clause**

- **This declares variables in its list to be private to each thread**
- **Format**
  - ✓ **private (list)**

Eg:

```
int B = 10;
#pragma omp parallel private(B)
B = … ;
```

- **A private un-initialised copy of B is created before the parallel region begins**
- **B value is not the same within the parallel region as outside**

Attribute Scoping contd..

❖ **firstprivate clause**

- **Format**
  - ✓ **first private (list)**
  - ✓ **Eg:**

```
int B;
B = 10;
#pragma omp parallel firstprivate(B)
B = B + … ;
```

- **A private initialized copy of B is created before the parallel region begins**
- **The copy of each thread gets the same value**

❖ SHARED Clause

✓ A shared variable exists in only one memory location and all threads can read or write to that address

- Format

✓ shared (list)

❖ DEFAULT Clause

✓ Specify default scope for all variables in the lexical extent.

- Format

✓ default (shared | none)

❖ LASTPRIVATE Clause

✓ Value from the last loop iteration assigned the original variable object.

- Format

✓ lastprivate (list)

❖ COPYIN Clause

- ✓initialized with value from master thread.
- ✓Used for threadprivate variables

- Format
  - ✓ copyin (list)

❖ COPYPRIVATE Clause

- ✓Used to broadcast values of single thread to all instances of the private variables
- ✓Associated with the SINGLE directive

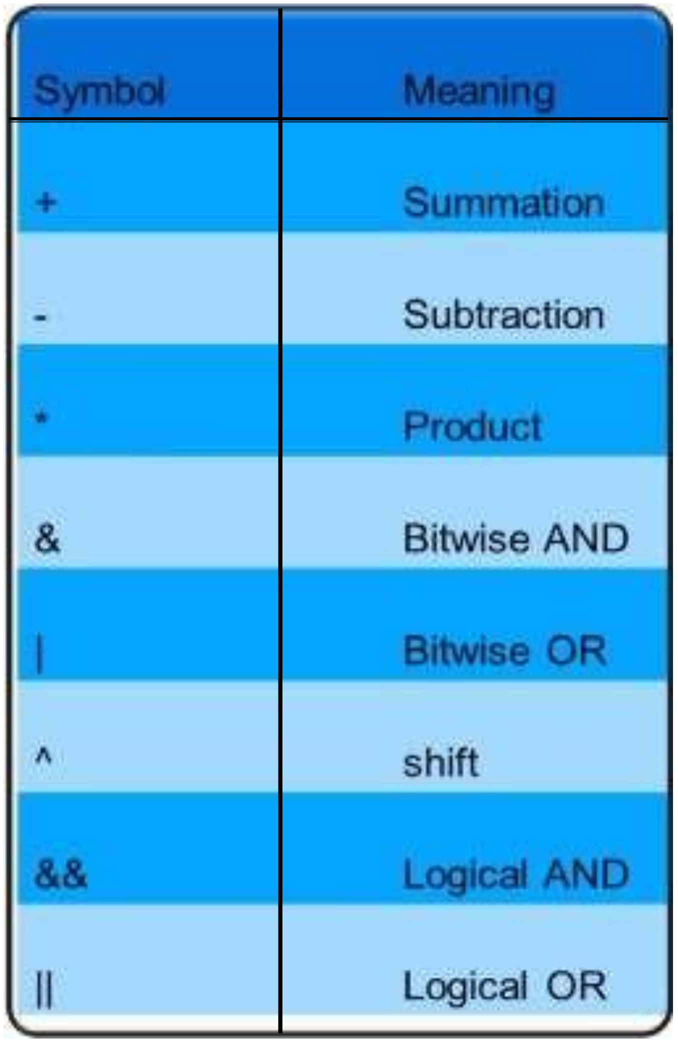
- Format
  - ✓ copyprivate (list)

❖ REDUCTION Clause

✓ Variables which needed to be shared & modified by all the processors

- **Format**
  - ✓ reduction (operator: list)
- **Example**

```
total = 0.0;
# pragma omp parallel for private ( i, p ) /
   shared ( n, x ) reduction ( +: total )

   for ( i = 0; i < n; i++ )
   {
          p = ( ( x[i] - 7 ) * x[i] + 4 ) * x[i] - 83;
          total = total + p;
    }
```



Symbol	Meaning
+	Summation
-	Subtraction
*	Product
&	Bitwise AND
|	Bitwise OR
^	shift
&&	Logical AND
||	Logical OR

Parallel Directive...

❖ **How Many Threads?**

- **The number of threads in a parallel region is determined by the following factors, in order of precedence:**
  - ✓ **Evaluation of the IF clause**
  - ✓ **Setting of the NUM_THREADS clause**
  - ✓ **Use of the omp_set_num_threads() library function**
  - ✓ **Setting of the OMP_NUM_THREADS environment variable**
  - ✓ **Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).**
- **Threads are numbered from 0 (master thread) to N-1**

```
#include <omp.h>

void subdomain(float *x, int istart, int ipoints)
{
  int i;

  for (i = 0; i < ipoints; i++)
      x[istart+i] = 123.456;
}

void sub(float *x, int npoints)
{
    int iam, nt, ipoints, istart;

#pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt =  omp_get_num_threads();
        ipoints = npoints / nt;    /* size of partition */
        istart = iam * ipoints;  /* starting array index */
        if (iam == nt-1)     /* last thread may do more */
          ipoints = npoints - istart;
        subdomain(x, istart, ipoints);
    }
}

int main()
{
    float array[10000];

    sub(array, 10000);

    return 0;
}
```

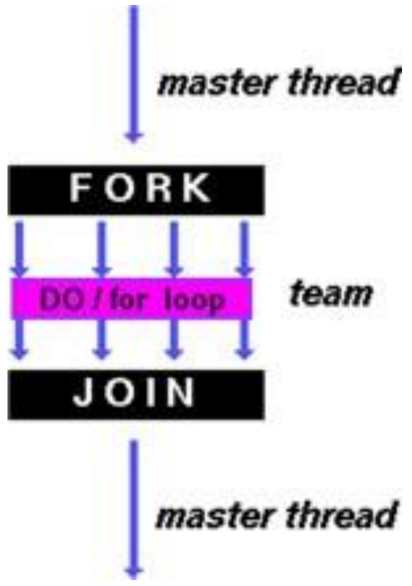
III. Work-Sharing Constructs

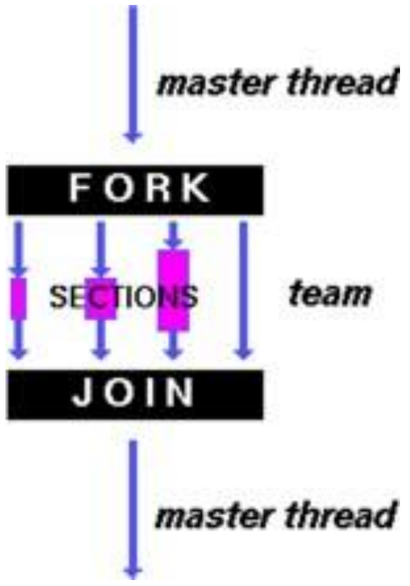
- ✓divides execution of code region among members of the team.
- ✓Work-sharing constructs do not launch new threads

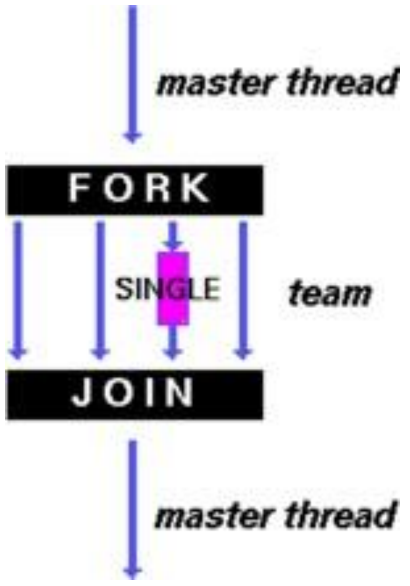
- Restrictions
  - ✓Must be enclosed within a parallel region.
  - ✓Work is distributed among the threads
  - ✓Encountered by all threads
  - ✓Does not launch new set of threads

Types of Work Sharing

- **FOR - data parallelism**
- **SECTIONS - functional parallelism**
- **SINGLE - serializes a section of code**







Work Sharing construct

❖ **for Directive**

- **for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team**
- **Format**

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

Restrictions for loop

for (index = start ; index < end ; increment_expr)

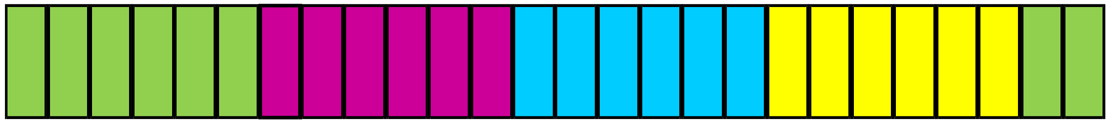
it must be possible to determine the number of loop iterations before execution

- ✓ **no while loops**
- ✓ **no variations of for loops where the start and end values change.**
- ✓ **increment must be the same each iteration**
- ✓ **all loop iterations must be done**
- ✓ **loop must be a block with single entry and single exit**
- ✓ **no break or goto**

```
for( i = 0, i< n, i++)
if (x[i]>maxval) goto 100;   //not parallelizable
```

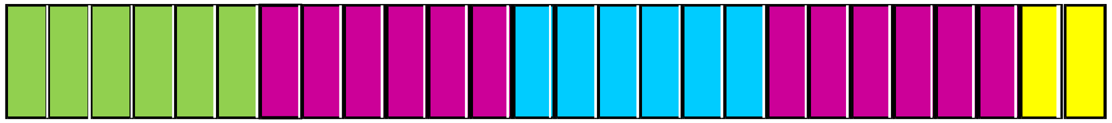
Contd..

- **Clauses**
  - ✓ **SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.**
    - **STATIC**



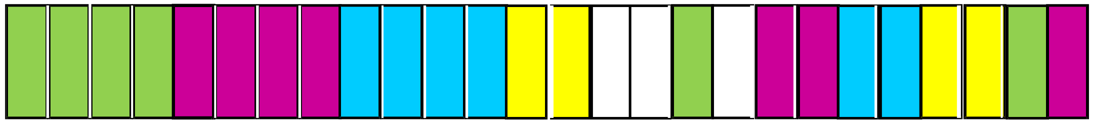
SCHEDULE(STATIC,6)
26 iter on 4 processors

    - **DYNAMIC**



SCHEDULE(DYNAMIC,6)
26 iter on 4 processors

    - **GUIDED**



SCHEDULE(GUIDED,4)
26 iter on 4 processors

    - **RUNTIME - determined by an environment variable OMP_SCHEDULE**

- **Clauses**
  - ✓ **NO WAIT / nowait: Threads do not synchronize at the end of the parallel loop.**
  - ✓ **ORDERED: Specifies that the iterations of the loop must be executed as they would be in a serial program**

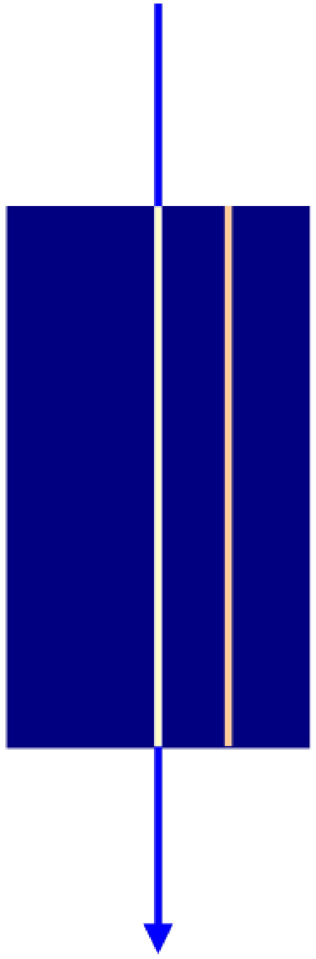
❖ SECTION Directive

- Each SECTION is executed once by a thread in the team
- Format

```
#pragma omp sections [clause ...] newline
                                  private (list)
                                  firstprivate (list)
                                  lastprivate (list)
                                  reduction (operator: list)
                                  nowait
{
        #pragma omp section newline
                    structured_block
        #pragma omp section newline
                    structured_block
}
```

- Clauses
  - ✓ NOWAIT: implied barrier exists at the end of a SECTIONS directive, unless this clause is used.
- Restriction
  - ✓ It is illegal to branch into or out of section blocks.
  - ✓ SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive

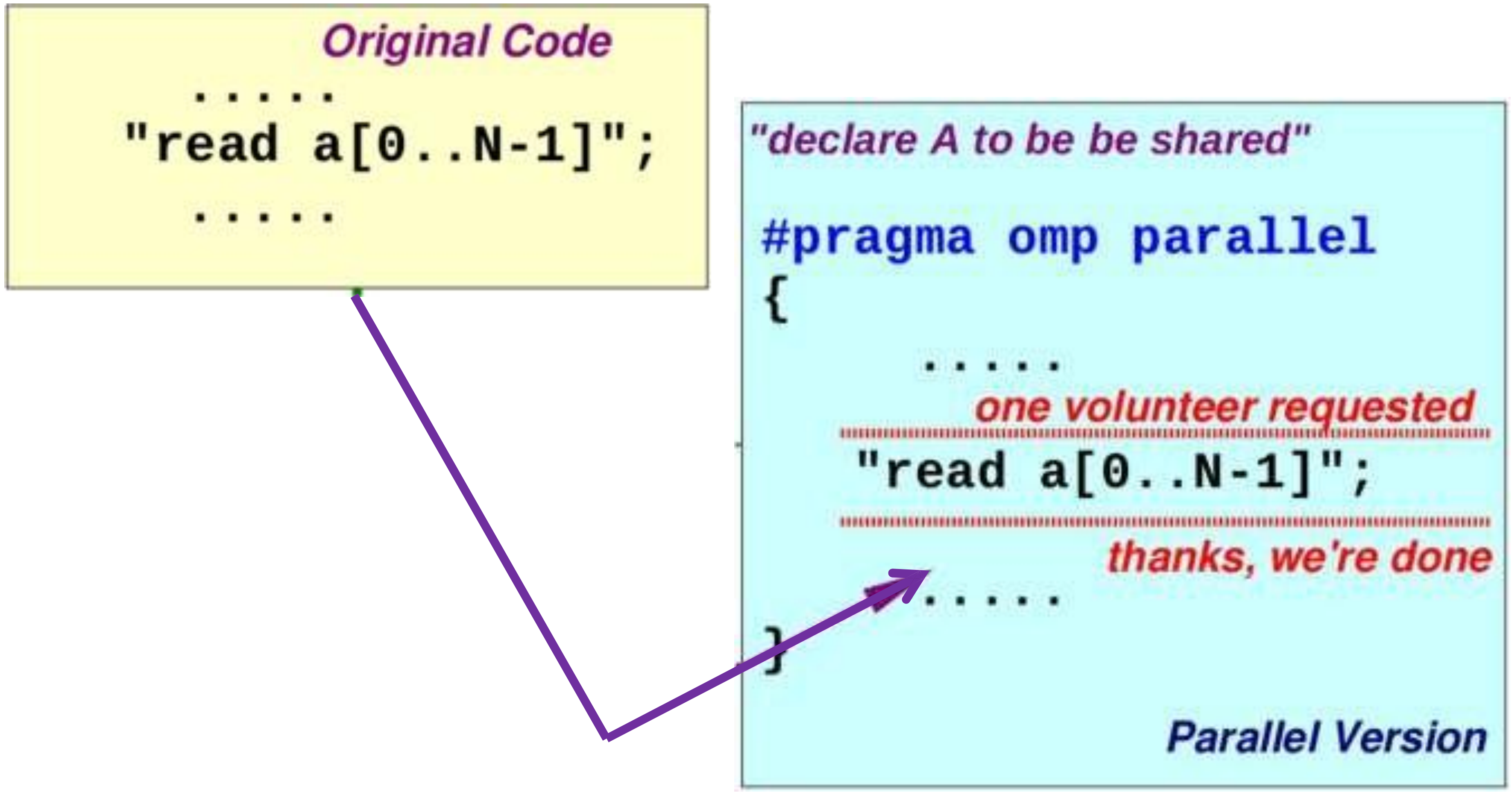
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
       for (i=0; i<n-1; i++)
           b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
       for (i=0; i<n; i++)
           d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```



❖ Single Directive

- 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)
- Format

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



```
    .....
"read a[0..N-1]";
    .....
```

```
#pragma omp parallel
{
        .....
    "read a[0..N-1]";
        .....
}
```

❖ Combined Parallel Work-Sharing Constructs

- These directives behave identical to individual parallel directives
- Types
  - ✓ parallel for
  - ✓ parallel sections

Combined constructs

```
#pragma omp parallel
#pragma omp for
   for (...)
```

```
#pragma omp parallel for
for (....)
```

Single PARALLEL loop

```
!$omp parallel
!$omp do
        ...
!$omp end do
!$omp end parallel
```

```
!$omp parallel do
        ...
!$omp end parallel do
```

Single WORKSHARE loop

```
!$omp parallel
!$omp workshare
        ...
!$omp end workshare
!$omp end parallel
```

```
!$omp parallel workshare
        ...
!$omp end parallel workshare
```

```
#pragma omp parallel
#pragma omp sections
{ ...}
```

```
#pragma omp parallel sections
{ ... }
```

Single PARALLEL sections

```
!$omp parallel
!$omp sections
        ...
!$omp end sections
!$omp end parallel
```

```
!$omp parallel sections
        ...
!$omp end parallel sections
```

Special Directive

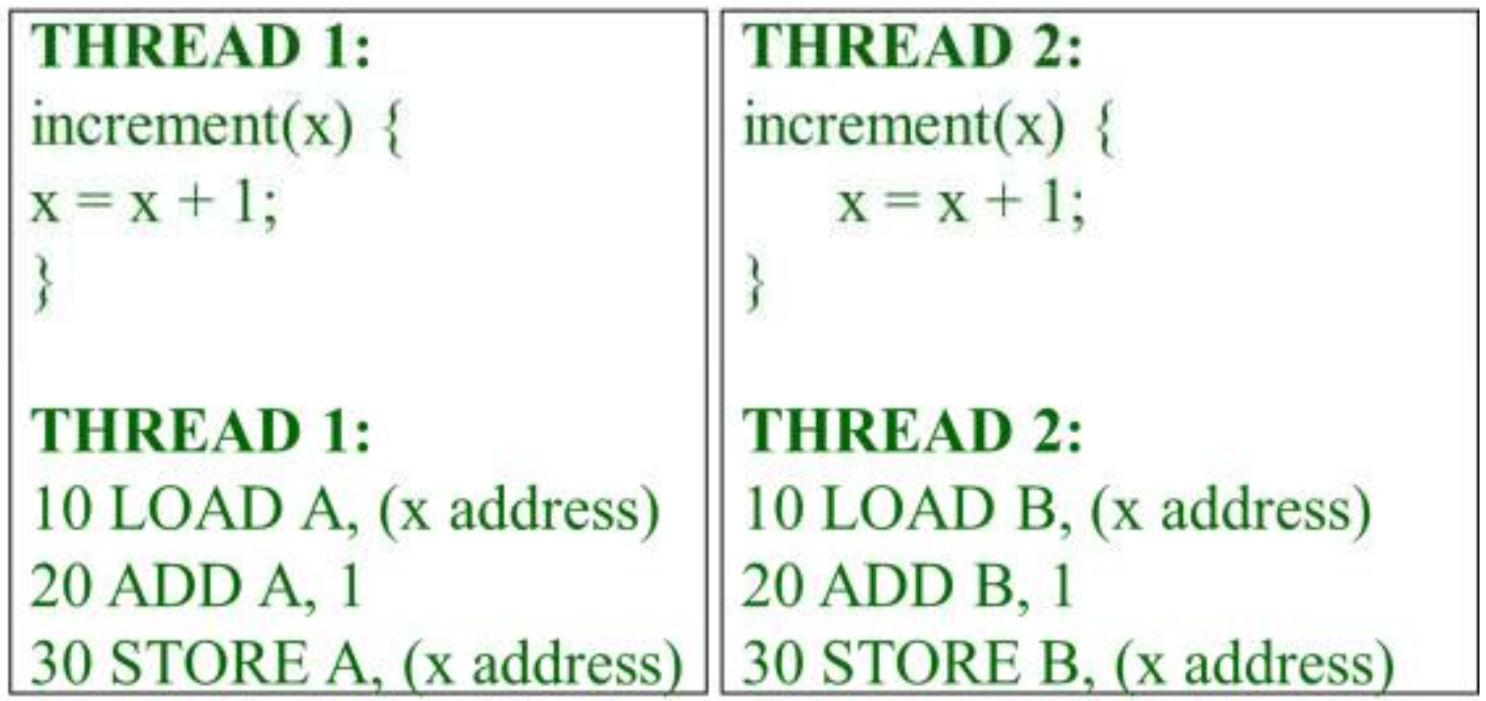
❖ **THREADPRIVATE Directive**

- **THREADPRIVATE variables differ from PRIVATE variables because they are able to persist between different parallel sections of a code.**
- **Format**
  - ✓ **#pragma omp threadprivate (list)**

Synchronization

IV. Synchronization Constructs

- **Two threads on two different processors are both trying to increment a variable x at the same time (assume x is initially 0):**



```
THREAD 1:
increment(x) {
x = x + 1;
}

THREAD 1:
10 LOAD A, (x address)
20 ADD A, 1
30 STORE A, (x address)
```

```
THREAD 2:
increment(x) {
    x = x + 1;
}

THREAD 2:
10 LOAD B, (x address)
20 ADD B, 1
30 STORE B, (x address)
```

❖ Synchronization Directives

- MASTER Directive
- CRITICAL Directive
- BARRIER Directive
- ATOMIC Directive
- FLUSH Directive
- ORDERED Directive

Synchronization Construct

❖ **MASTER Directive**

- **executed only by master thread of the team.**
- **All other threads on the team skip this section of code**
- **There is no implied barrier associated with this directive**
- **Format**

```
#pragma omp master newline
                                   structured_block
```

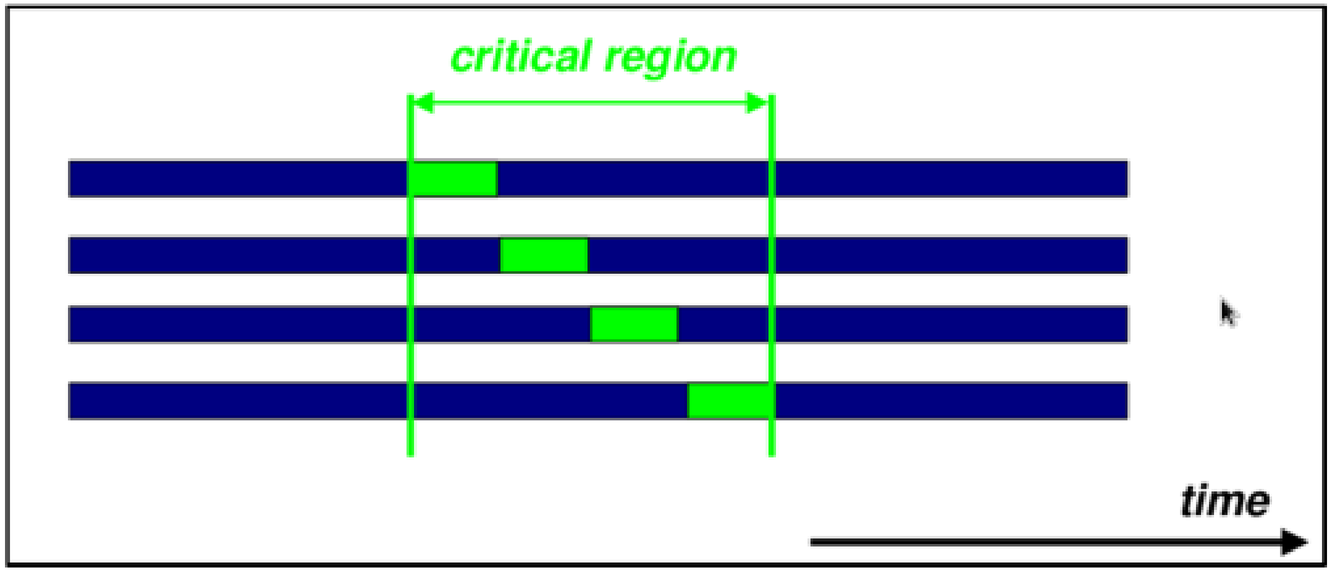
❖ CRITICAL Directive

✓The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.

- Format

#pragma omp critical [name] newline structured_block

The optional name enables multiple different CRITICAL regions to exist:



Synchronization Construct

❖ **BARRIER Directive**

- ✓ **On reaching BARRIER directive , 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.**

- **Format**
  - ✓ **#pragma omp barrier newline**

Barrier Sample

```
#pragma omp parallel for
    for (i=0; i < N; i++)

        a[i] = b[i] + c[i];
#pragma omp barrier
#pragma omp parallel for
    for (i=0; i < M; i++)

        d[i] = a[i] + b[i];
```

❖ Atomic Directive

- specifies that a specific memory location must be updated atomically
- Avoids simultaneous update from many threads

- Format

```
#pragma omp atomic newline
          statement_expression
```

❖ FLUSH Directive

- Identifies a synchronization point at which the implementation must provide a consistent view of memory
- Format

```
#pragma omp flush (list) newline
```

-- The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables.

- **The FLUSH directive is implied for the directives shown in the table below**
  - ✓ **Barrier**
  - ✓ **Critical**
  - ✓ **For (upon exiting)**
  - ✓ **Parallel (upon exiting)**
  - ✓ **Sections (upon exiting)**
  - ✓ **Single (upon exiting)**
  - ✓ **Ordered (upon entry to and exit from)**

❖ Ordered Directive

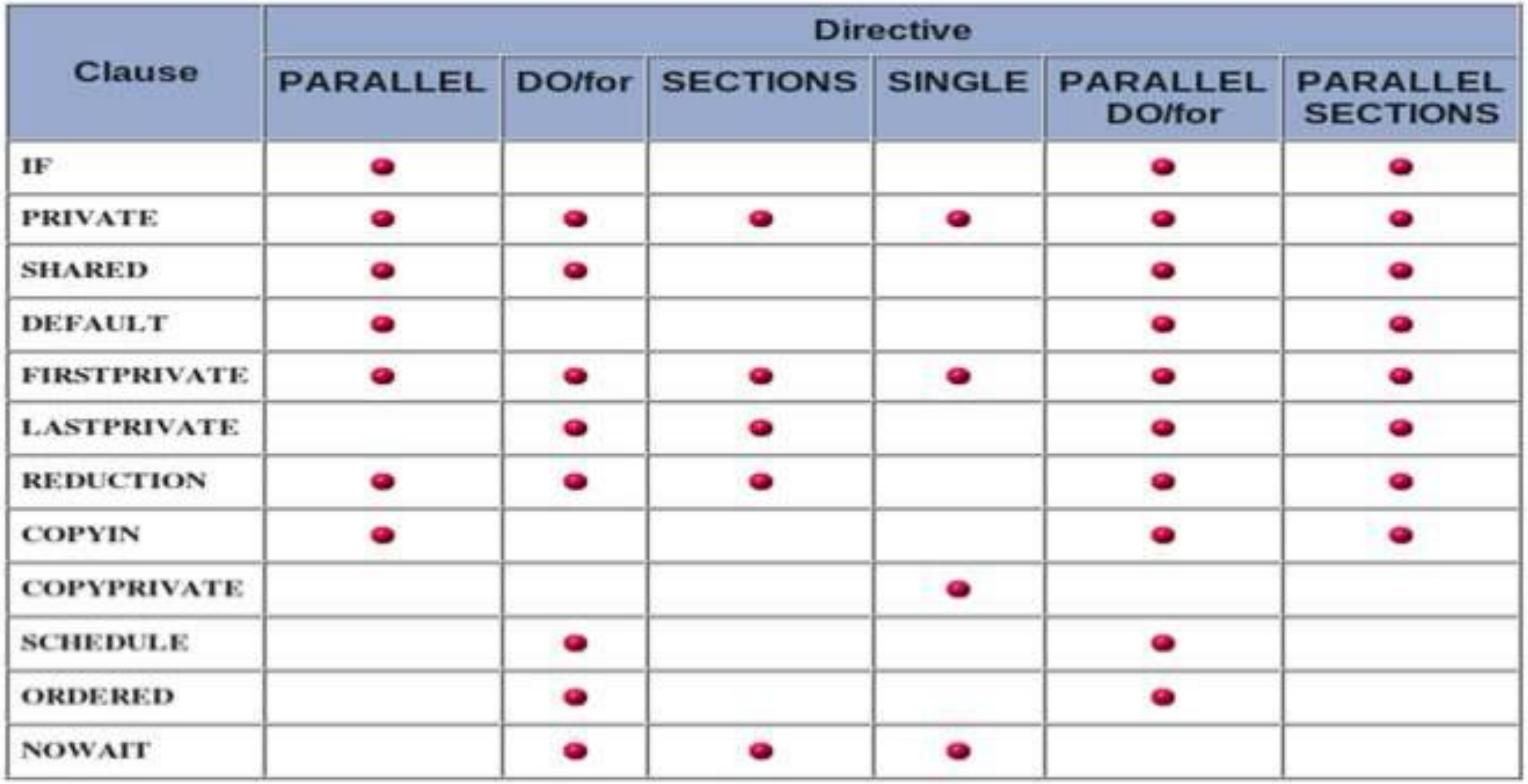
- **#pragma omp ordered**

  ✓ **Must appear within for or parallel for directive**

  ✓ **Only 1 thread at a time is allowed into an ordered section**

  ✓ **The thread executes the iterations in the same order as the iterations are executed in sequential loop**

Clauses /Directive Summary



Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

- **The rest of them do not have clauses**

Runtime Libraries

- **Execution environment routines that can be used to control and to query the parallel execution environment**
- **Lock routines that can be used to synchronize access to data**

Runtime Libraries

❖ **OMP_SET_NUM_THREADS**

- **omp_set_num_threads routine affects the number of threads to be used for subsequent parallel regions**
  - ✓ **C/C++ : void omp_set_num_threads(int num_threads);**

❖ **OMP_GET_NUM_THREADS**

- **returns the number of threads in the current team.**
  - ✓ **C/C++ : int omp_get_num_threads(void);**

Runtime Libraries

- **OMP_GET_THREAD_NUM**
  - **Returns the thread ID of the thread**

```
#include <omp.h>
int omp_get_thread_num()
```

- **OMP_GET_NUM_PROCS**
  - **To get the number of processors**

```
#include <omp.h>
int omp_get_num_procs()
```

- **OMP_IN_PARALLEL**
  - **determine if the section of code which is executing is parallel or not.**

```
#include <omp.h>
int omp_in_parallel()
```

❖ OMP_SET_DYNAMIC

- Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions.

```
#include <omp.h>
void omp_set_dynamic(int val)
```

- Remarks :
  - ✓The number of threads will never exceed the value set by omp_set_num_threads or by OMP_NUM_THREADS.
  - ✓ Use omp_get_dynamic to display the current setting of omp_set_dynamic.
  - ✓ The setting for omp_set_dynamic will override the setting of the OMP_DYNAMIC environment variable.

- **OMP_GET_DYNAMIC**
  - **Determine if dynamic thread adjustment is enabled or not.**

```
#include <omp.h>
int omp_get_dynamic()
```

- **OMP_SET_NESTED**
  - **Used to enable or disable nested parallelism**

```
#include <omp.h>
void omp_set_nested(int nested)
```

- **OMP_GET_NESTED**

```
#include <omp.h>
int omp_get_nested ()
```

Environment Variables

- OMP_SCHEDULE
  - setenv OMP_SCHEDULE "guided"
  - setenv OMP_SCHEDULE "dynamic"
- OMP_NUM_THREADS
  - setenv OMP_NUM_THREADS 8
- OMP_DYNAMIC
  - setenv OMP_DYNAMIC TRUE
- OMP_NESTED
  - setenv OMP_NESTED TRUE

Programming Tips

- Start from an optimized serial version.
- Gradually add OpenMP, check progress, add barriers.
  - ✓ Use profilers to understand code
- Decide which loop to parallelize - outer loop or loop permutation, fusion, exchange or collapse.
- Adjust environment variables.
- Minimize shared and barriers , maximize private.
- Minimize parallel constructs, if possible use combined constructs.
- Take advantage of debugging tools: gdb, totalview, DDT, etc.

Look out for - correctness

- ✓ Access to shared variables not protected
- ✓ Read of shared variable without obeying the memory model
- ✓ Forget to mark private variables as such
- ✓ Use of ordered clause without ordered construct
- ✓ Declare loop variable in for-construct as shared
- ✓ Try to change the number of threads in a parallel region, after it has been started already
- ✓ Attempt to change loop variable while in #pragma omp for

Directive Bindings

❖ **Directive Bindings**

- ✓ **The for, SECTIONS, SINGLE, MASTER and BARRIER directives bind to the dynamically enclosing PARALLEL.**
- ✓ **The ORDERED directive binds to the dynamically enclosing for.**
- ✓ **ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.**
- ✓ **The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.**
- ✓ **A directive can never bind to any directive outside the closest enclosing PARALLEL.**

Directive Nesting

- ✓ A worksharing region may not be closely nested inside a worksharing, critical, ordered, atomic, or master region.
- ✓ A barrier region may not be closely nested inside a worksharing, critical, ordered, atomic, or master region.
- ✓ A master region may not be closely nested inside a worksharing, atomic, or explicit task region.
- ✓ An ordered region may not be closely nested inside a critical, atomic, or explicit task region.
- ✓ An ordered region must be closely nested inside a loop region (or parallel loop region) with an ordered clause.
- ✓ A critical region may not be nested (closely or otherwise) inside a critical region with the same name. Note that this restriction is not sufficient to prevent deadlock.
- ✓ parallel, flush, critical, atomic, taskyield, and explicit task regions may not be closely nested inside an atomic region.

Dependencies

❖ **True Dependence**

- **Statements S1, S2**
- **S2 has a true dependence on S1 iff S2 reads a value written by S1**

❖ **Anti Dependence**

- **S2 has an anti-dependence on S1 iff S2 writes a value read by S1**

❖ **Output Dependence**

- **S2 has an output dependence on S1 iff S2 writes a variable written by S1.**

there are no dependences between S1 and S2

- ✓true dependences
- ✓anti-dependences
- ✓output dependences

- Some dependences can be removed.

❖ Loop Dependencies

```
for (i=0; i<10;i++)
    a(i) = a(i) + a(i – 1)
```

A simple loop with a data dependence.

- whenever there is a dependence between two statements on some location, we cannot execute the statements in parallel.
  - ✓ it would cause a data race.
  - ✓ parallel program may not produce the same results as an equivalent serial program.

- **Example**

```
for(i=0; i<100; i++)
a[i] = a[i] + 100;
```

- **Example**

```
for( i=0; i<100; i++ )
a[i] = f(a[i-1]);
```

- **Points to ponder**
  - ✓ Statement order must not matter.
  - ✓ Statements must not have dependences.
  - ✓ Some dependences can be removed.
  - ✓ Some dependences may not be obvious.

Performance

- **Coverage**
  - ✓percentage of a program that is parallel.
- **granularity**
  - ✓ extent to which a program is broken down into small parts.
- **load balancing**
  - ✓ how evenly balanced the work load is.
  - ✓ loop scheduling determines how iterations of a parallel loop are assigned to threads
- **locality and synchronization**
  - ✓cost to communicate information between different processors on the underlying system.
  - ✓ need to understand machine architecture

Performance considerations

❖ **Coping with parallel overhead**

- **best to parallelize the loop that is as close as possible to being outermost**
  - ✓**because of parallel overhead incurred each time we reach a parallel loop**

- **Example**

```
#omp parallel for
for (i = 1; i<n; i++)
    for ( j = 2; j<n ;j++)
        a[i, j] = a[i, j] + a[i, j–1]
```

Contd..

- **If data dependencies exist, the outermost loop in a nest may not be parallelizable**

```
for ( j = 2; j<n ;j++)   // Not parallelizable  - why?.
        for (i = 1; i<n; i++)       //Parallelizable.
          a[i, j] = a[i, j] + a[i, j–1]
```

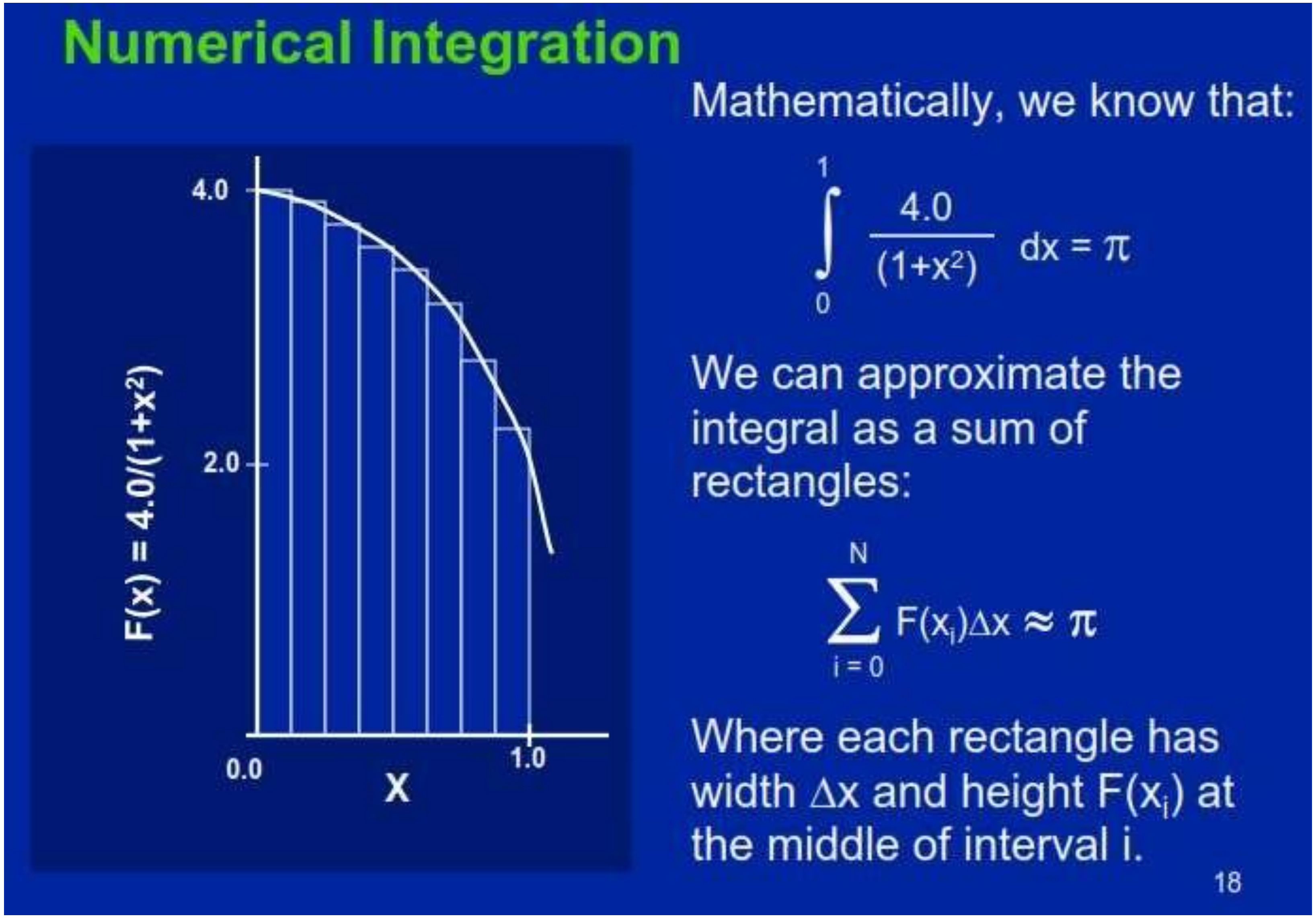
- **Solution**
  - ✓ **loop interchange that swaps the positions of inner and outer loops**
  - ✓ **Tradeoffs**
    - **but the transformed loop nest has worse utilization of the memory cache.**
    - **transformations may involve a tradeoff - they improve one aspect of performance but hurt another aspect**

Considerations - performance

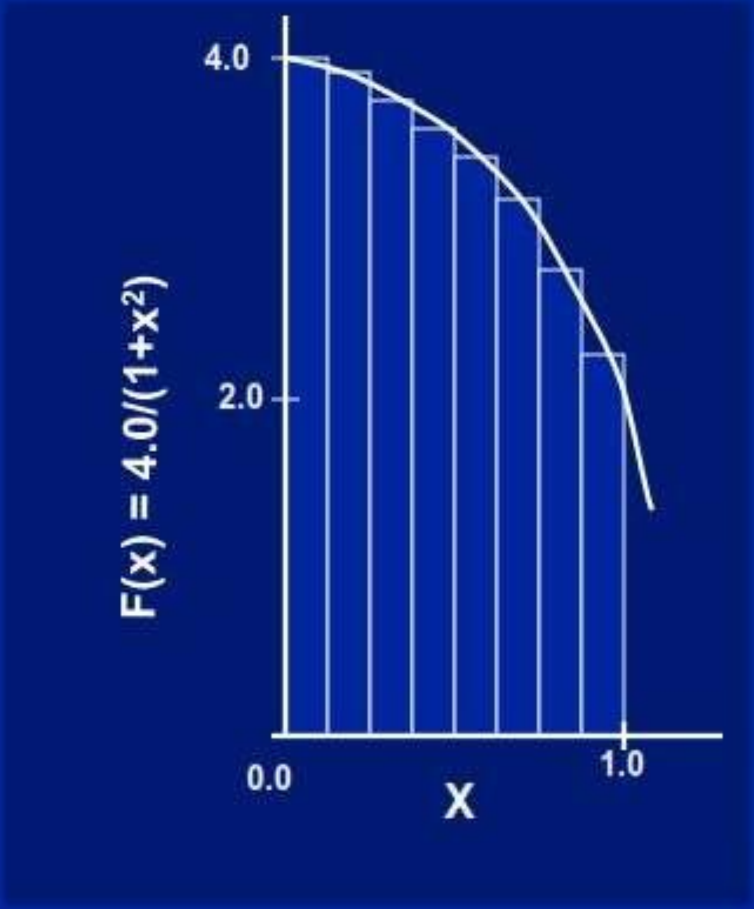
❖ **What not to do ?**

- **Use of critical when atomic would be sufficient**
- **Put too much work inside critical region**
- **Use of orphaned construct outside parallel region**
- **Use of unnecessary flush**
- **Use of unnecessary critical**

PI Calculation



Numerical Integration



Mathematically, we know that:

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

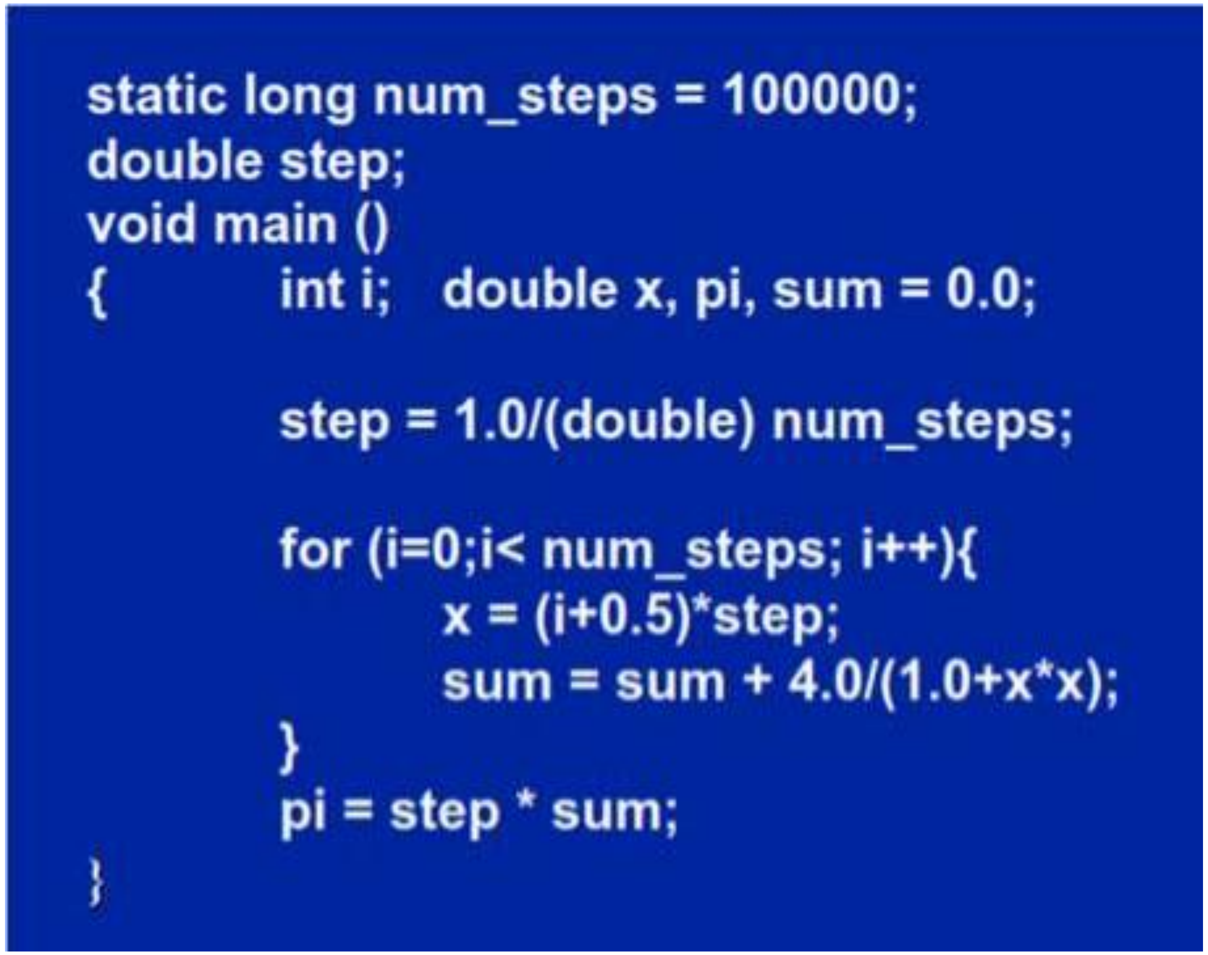
We can approximate the integral as a sum of rectangles:

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

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

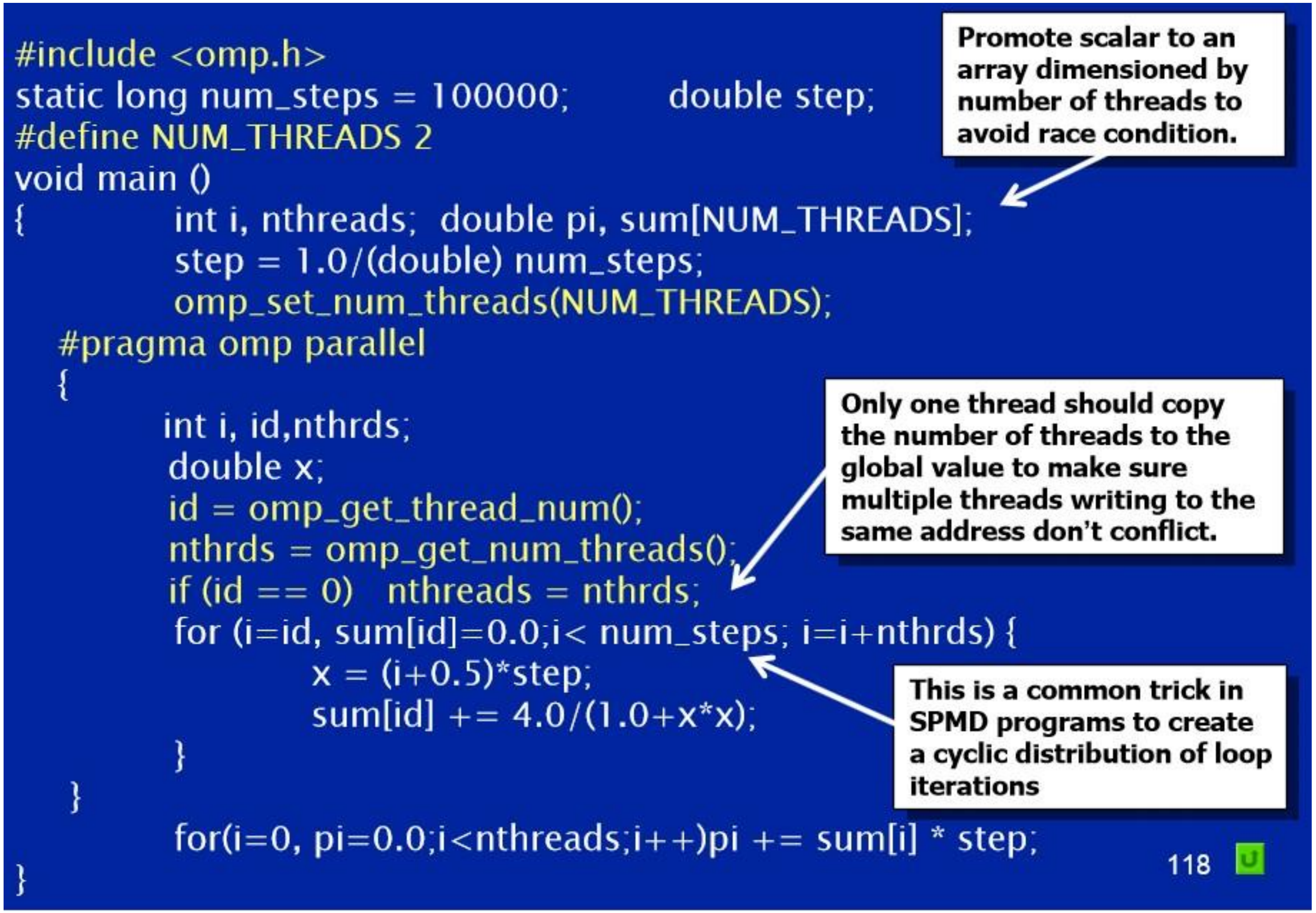
18

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

Simple PI



```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{            int i, nthreads;  double pi, sum[NUM_THREADS];
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
            int i, id,nthrds;
             double x;
             id = omp_get_thread_num();
             nthrds = omp_get_num_threads();
             if (id == 0)    nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                        x = (i+0.5)*step;
                        sum[id] += 4.0/(1.0+x*x);
             }
     }
             for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

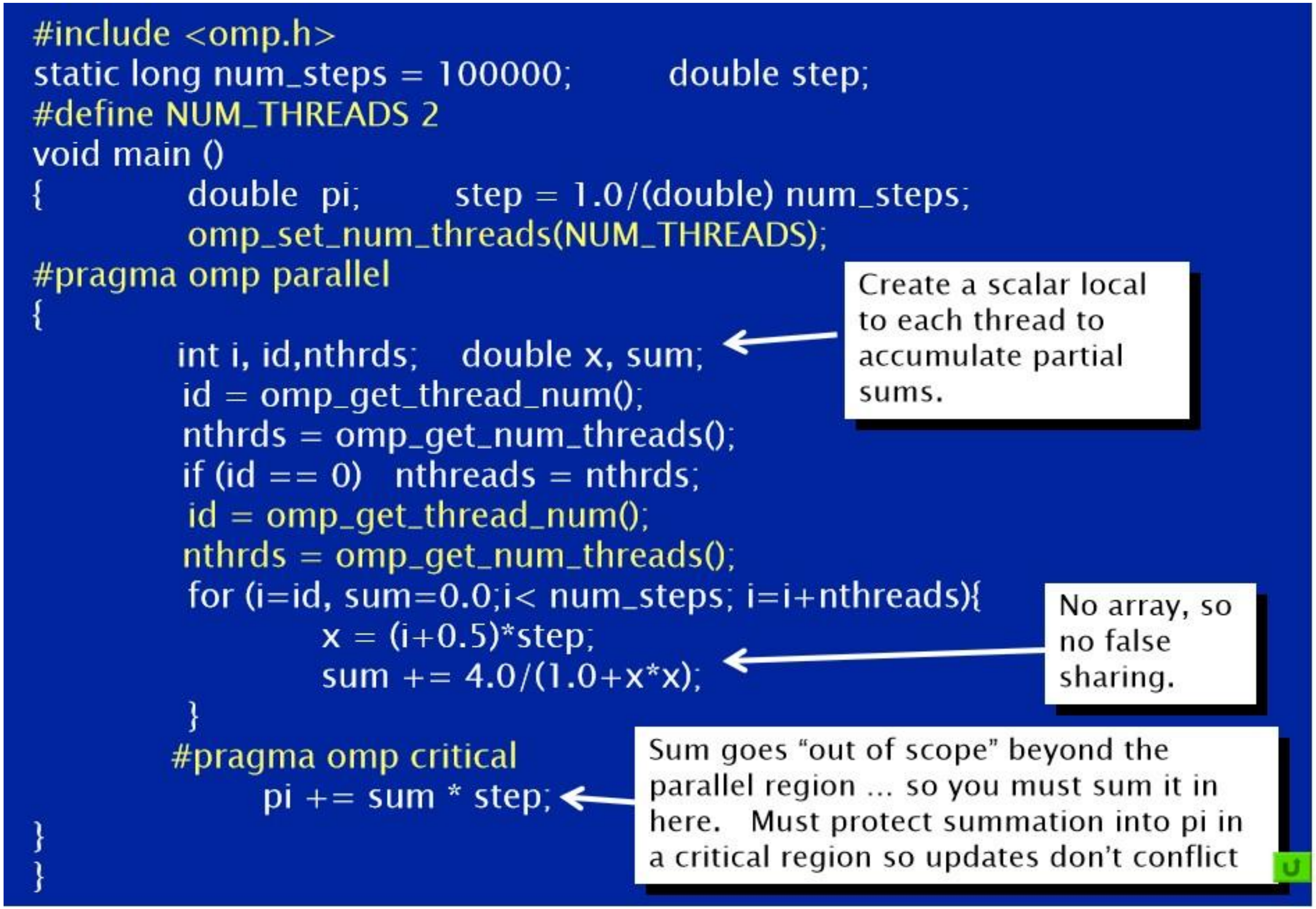
Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

118

PI with critical



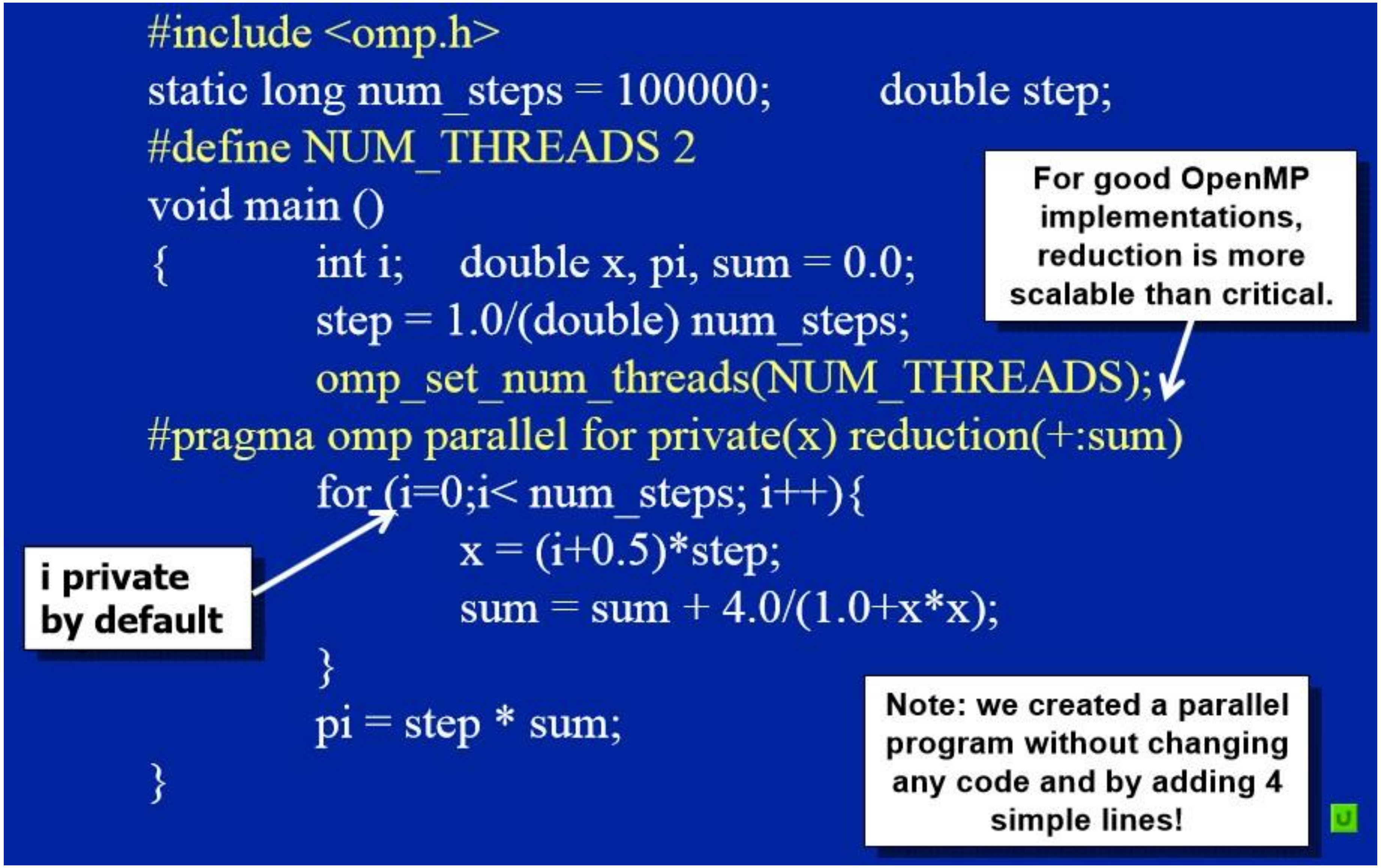
```
#include <omp.h>
static long num_steps = 100000;          double step;
#define NUM_THREADS 2
void main ()
{           double  pi;         step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
          int i, id,nthrds;    double x, sum;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
            id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
            for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                      x = (i+0.5)*step;
                      sum += 4.0/(1.0+x*x);
            }
          #pragma omp critical
                pi += sum * step;
}
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes "out of scope" beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don't conflict

PI with reduction



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

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

Note: we created a parallel program without changing any code and by adding 4 simple lines!

THANK YOU !!

Any Questions?