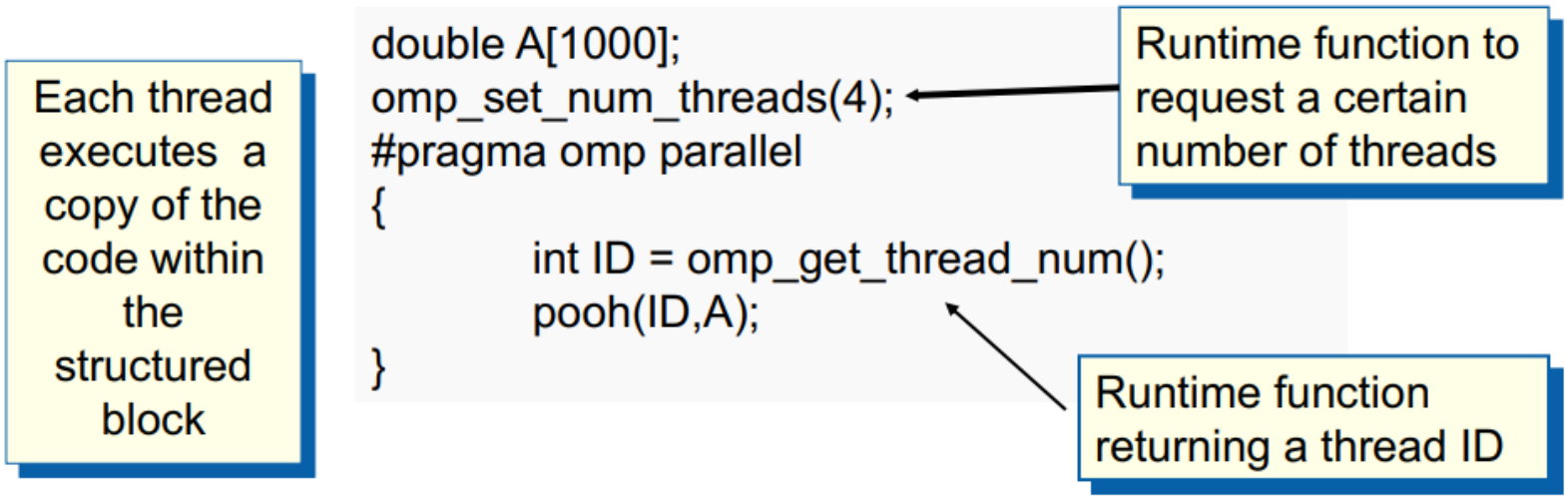
Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, to create a 4 thread Parallel region:



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

Thread creation: How many threads did you actually get?

- Request a number of threads with omp_set_num_threads()
- The number requested may not be the number you actually get.
  - An implementation may silently give you fewer threads than you requested.
  - Once a team of threads has launched, it will not be reduced.

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();
        int nthrds = omp_get_num_threads();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

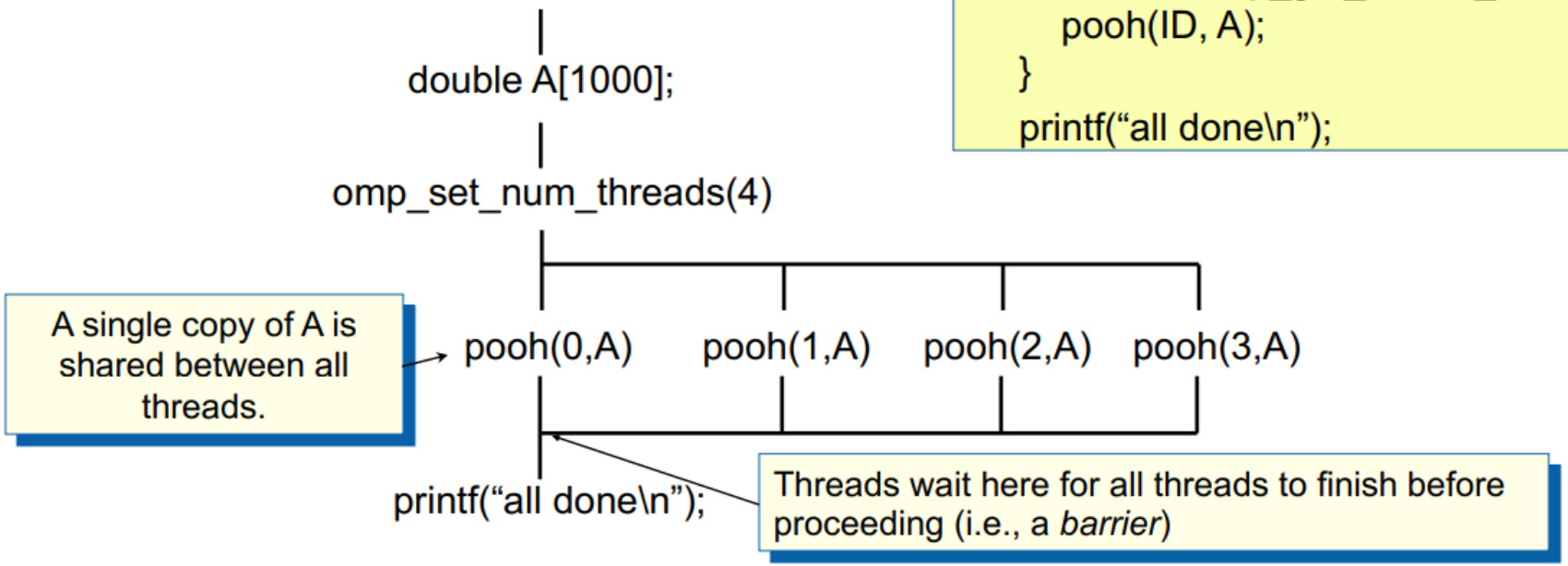
Runtime function to return actual number of threads in the team

- Each thread calls pooh(ID,A) for ID = 0 to nthrds-1

Thread Creation: Parallel Regions Example

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



OpenMP Components

Compiler Directives

A. **Parallel construct**
B. **Work Sharing**
C. **Synchronization**
D. **Data Scope**
   - **Private**
   - **Shared**
   - **First private**
   - **Last private**
   - **Reduction**

Runtime Library

- **Number of threads**
- **Thread ID**

Environment Variables

- **Number of threads**
- **Scheduling Type**

OpenMP Compiler Directives

- Spawning a parallel region
- Used to divide blocks of code among threads.
- Distributing loop iterations among threads.
- Serializing sections of code.
- Synchronization of work among threads.

Runtime Library Routines

- Setting and querying the number of threads.
- Querying unique thread ID.
- To check whether inside parallel region.

Environment Variables

- Setting the number of threads.
  - Export OMP_NUM_THREADS=8
- Specifying how loop iterations are divided.
- Binding threads to processors

1. OpenMP Compiler Directives

- Syntax:
  - #pragma omp directive-name [clause1 clause2..] new-line
- Example:
  - #pragma omp parallel
  - #pragma omp parallel private(i,j)
- Example Directives
  - Parallel
  - for/do
  - Sections
  - Single
  - Critical
  - Barrier

OpenMP Code Structure

- #pragma omp <directive-name> [clause,clause,..] new-line\
- General Rules
  - Case sensitive
  - Compiler Directives follow C/C++ standards
  - Only one directive name to be specified per directive

Parallel Directive

- Purpose: creating a team of threads in the parallel region.
- Format:

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

- 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
  - Only a single IF clause is permitted
  - Only a single NUM_THREADS clause is permitted

A. Parallel Directive

- The parallel construct forms a team of threads and starts parallel execution of a parallel region.
- A parallel region is a block of code that will be executed by multiple threads.

#pragma omp parallel [clause1 clause2...] newline

structured_block

Clauses:

if (scalar_expression)

private (list)

firstprivate (list)

num_threads (integer-expression)

C/C++ - General code Structure

```
#include <omp.h>
main () {

}       int var1, var2, var3;
        Serial code
        .
        .
        .

        Beginning of parallel section. Fork a team of threads.
        Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
{
        Parallel section executed by all threads
                .
        Other OpenMP directives
                .
        Run-time Library calls
                .
        All threads join master thread and disband

 }
Resume serial code
    .    .   .
}
```

Example: parallel construct

```
#include <omp.h>
main ()
{
int nthreads, tid;
/* Fork a team of threads with each thread having a private tid */
#pragma omp parallel private(tid)
{
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */ if
        (tid == 0)
        {
                nthreads = omp_get_num_threads();
                printf("Number of threads = %d\n", nthreads);
        }
  }  /* Implicit Barrier - All threads join master thread and terminate */
}
```

- 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.
- 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.

DIFFERENT WAYS OF THREAD CREATION

- The number of threads in a parallel region is determined by the following factors, in order of precedence (Low- High):
  1. Default number of Threads (number of cores)
  2. Setting of the OMP_NUM_THREADS environment variable

     export OMP_NUM_THREADS=2
  3. Use of the omp_set_num_threads() library function

     omp_set_num_threads(4);
  4. Setting of the NUM_THREADS clause

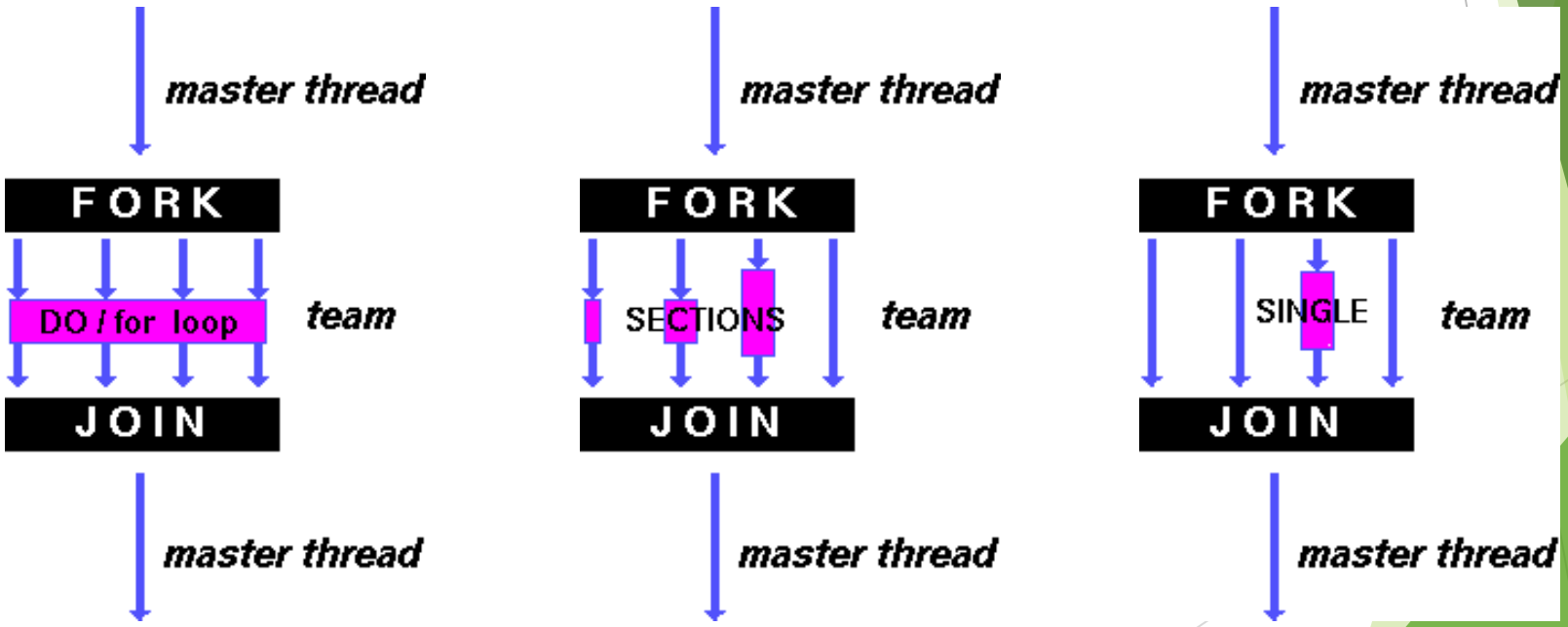
     #pragma omp parallel private(tid) num_threads(6)
- Implementation default - usually the number of cores.
- Threads are numbered from 0 (master thread) to N-1

B. Work Sharing Constructs

- Divides execution of code region among members of the team of threads.
- 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

Examples:

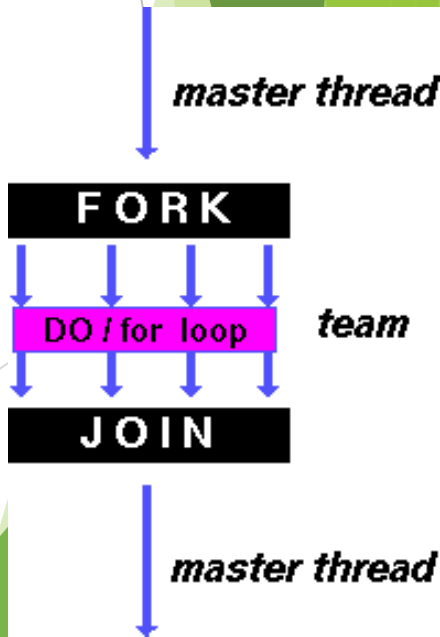
- For – data parallelism
- Section – functional parallelism
- single – serializes a section of code



FOR DIRECTIVE

- for directive divide the loop iterations among threads and execute in parallel.
- Purpose: For directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team
- Syntax:

```
#pragma omp for [clause1  clause2  ...] newline
 for_loop
```



Example: for construct

```
#include <omp.h>
#define N      1000

main ()
{
int i;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)          a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)
{
            #pragma omp for for
            (i=0; i < N; i++)
                        c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

RESTRICTIONS

- for (index = start ; index < end ; increment_expr)
  - It should be possible to determine the number of loop iterations before execution, to divide the iterations.
- No while loops
- No variations of for loops where the start and end values change
- increment must be same each iteration
- It is illegal to branch(goto) out of a loop associated with a for directive

If

- A clause to decide at run time if a parallel region should actually be executed in parallel (multiple threads) or just by the master thread:

- Syntax:

  If(logical expr)

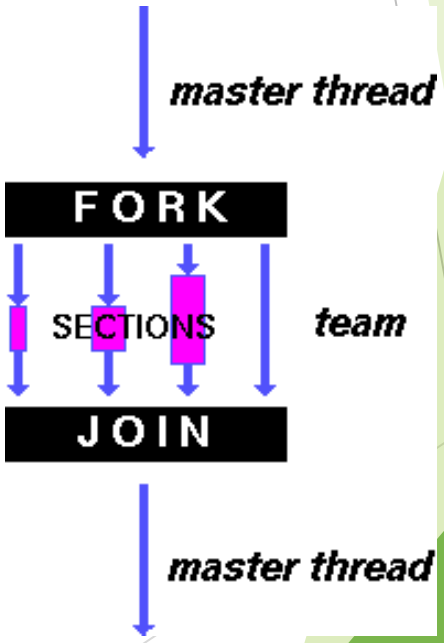
- Example:

  #pragma omp parallel if (n>100000)

SECTION DIRECTIVE

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

```
#pragma omp sections [clause1 clause2 newline
{
    #pragma omp section
                    structured_block
    #pragma omp     section  <newline>
                    structured_block
}
```



- Key points
  - NOWAIT : Implicit barrier at the end of sections directive, if nowait clause is present.
  - If “too many” sections, some threads execute more than one section (round-robin).
  - If “too few” sections, some threads are idle.
  - We don’t know in advance which thread will execute which section.
- Restrictions
  - It is illegal to branch into or out of section blocks
  - SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive

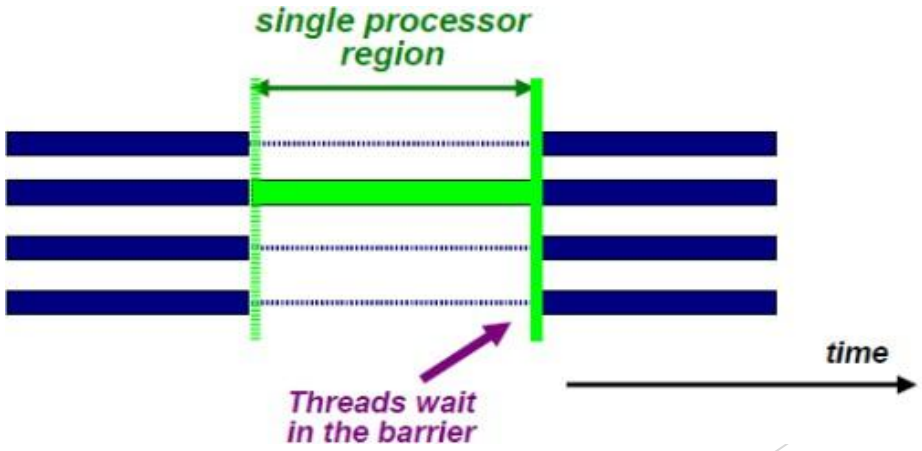
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

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

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

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

No need to wait to enter Or to exit from a section

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
- A barrier is implicitly set at the end of the single block (the barrier can be removed by the nowait clause)
- Syntax:

```
#pragma omp single [clause1 clause2 ...]
structured_block
```



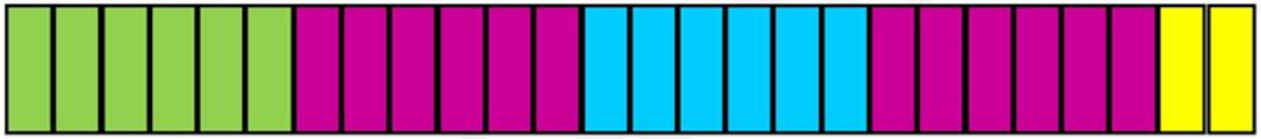
CLAUSES

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

- STATIC

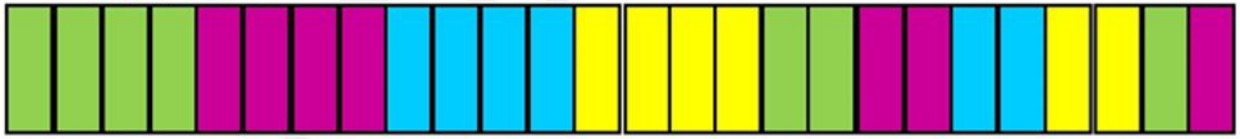
SCHEDULE(STATIC,6)
26 iter on 4 processors

- DYNAMIC



SCHEDULE(DYNAMIC,6)
26 iter on 4 processors

- GUIDED



SCHEDULE(GUIDED,4)
26 iter on 4 processors

- RUNTIME – determined by the environment variable OMP_SCHEDULE

SYNCHRONIZATION

- Synchronization is the task of ensuring that the multiple threads can safely share the resources.
- It is used to avoid the race condition

- Consider a case of 2 threads, both trying to increment a variable “x” at the same time (assume x = 0 initially)
- Problem: One possible execution sequence:
  - Thread 1 loads the value of x into register A.
  - Thread 2 loads the value of x into register A.
  - Thread 1 adds 1 to register A → x = x + 1 by thread1
  - Thread 2 adds 1 to register A → x = x + 1 by thread2
  - Thread 1 stores register A at location x
  - Thread 2 stores register A at location x
  - The resultant value of x will be 2, not 1 as it should be.

SYNCHRONIZATION

- Solution: To avoid a situation like this, the incrementing of x must be synchronized between the two threads to ensure that the correct result is produced.

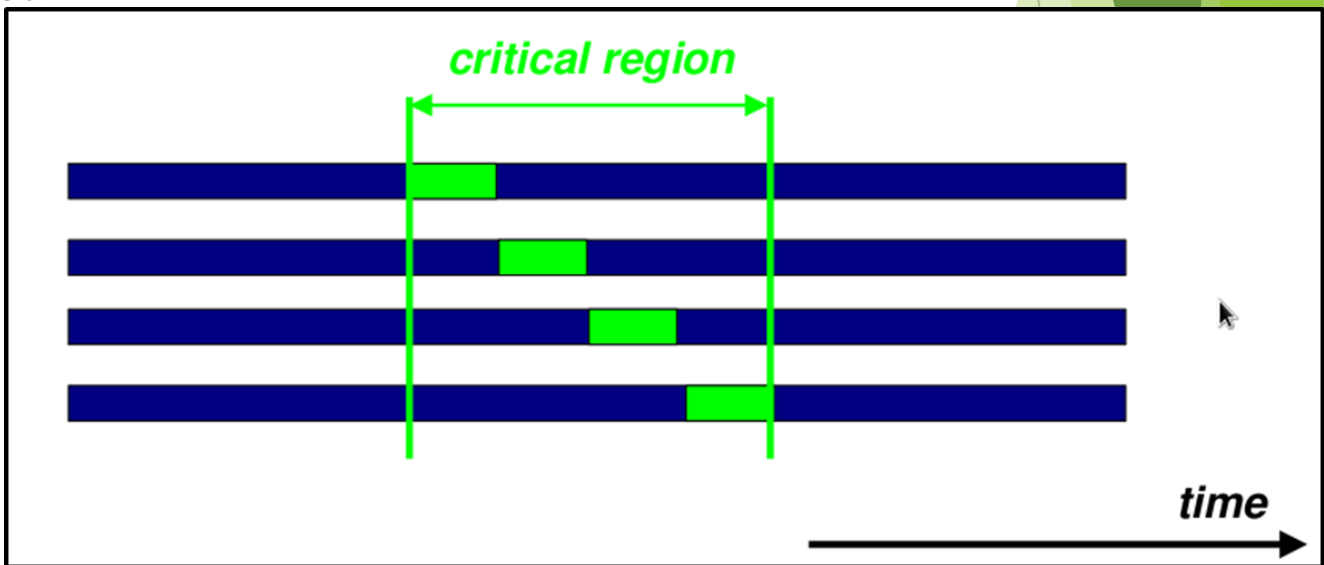
- OpenMP Synchronization Constructs:
  - CRITICAL Directive
  - ATOMIC Directive
  - BARRIER Directive
  - MASTER Directive

CRITICAL

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.
- Critical section prevents multiple threads from accessing a section of code at the same time.
- Only one active thread can update the code inside the critical region.
- Syntax:

```
#pragma omp critical
{ structured_block
}
```

The optional name enables multiple different CRITICAL regions to exist.



CRITICAL

- If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.
- The optional name enables multiple different CRITICAL regions to exist:
  - Names act as global identifiers. Different CRITICAL regions with the same name are treated as the same region.
  - All CRITICAL sections which are unnamed, are treated as the same section.

ATOMIC

- Specifies that a specific memory location must be updated atomically, by a single thread
- It is used to prevent race conditions
- Atomicity: We cannot split an operation – Example, A write operation.
- The directive applies only to a single, immediately following statement → Applies to only a single statement, allows only a limited number of expressions.
- Provides a mini critical section

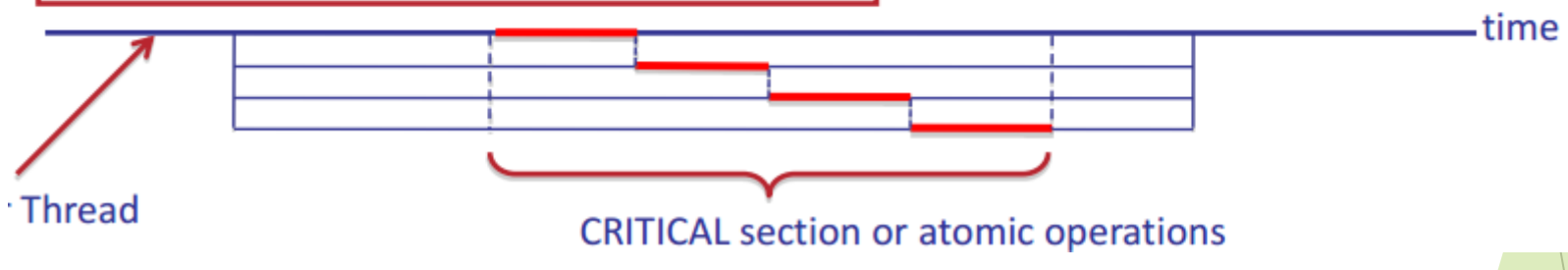
- Syntax:

```
#pragma omp atomic
{
    statement_expression
}
```

C.4 ATOMIC .VS. CRITICAL

```
#pragma omp parallel shared(sum,x,y)
  ...
#pragma omp critical
    {
      update(x);
      update(y);
      sum=sum+1;
    }
...
!$OMP END PARALLEL
```

```
#pragma omp parallel shared(sum)
...
     {
#pragma omp atomic
       sum=sum+1;
       ...
     }
```



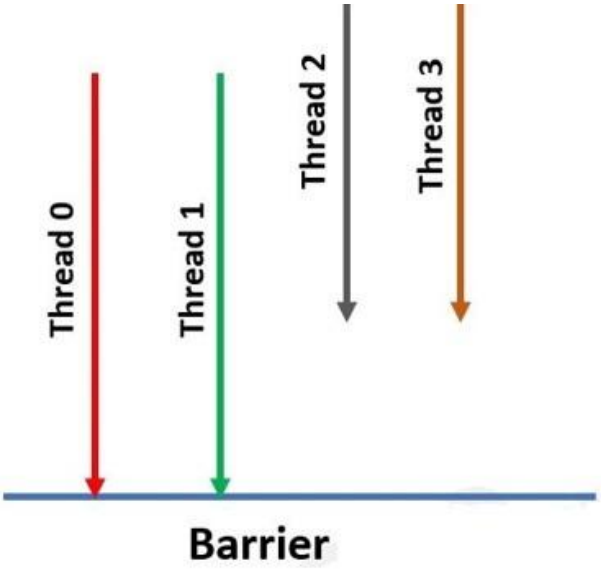
- **This Directive is very similar to the CRITICAL directive.**
- **Difference is that ATOMIC is only used for the update of a memory location.**
- **Sometimes ATOMIC is also referred to as a mini critical section.**

BARRIER

- 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. Synchronizes all threads in a team.
- Syntax:

#pragma omp barrier newline

```
#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];
```


Thread 0
Thread 1
Thread 2
Thread 3
Barrier


Thread 0
Thread 1
Thread 2
Thread 3
Barrier


Barrier
Thread 0
Thread 1
Thread 2
Thread 3

MASTER

- To execute a region 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
- Syntax:

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

DATA SCOPE ATTRIBUTES

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

- IF
- PRIVATE
- FIRSTPRIVATE
- SHARED

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

PRIVATE

- This declares variables in its list to be private to each thread
- Syntax:

  private (list)

  Eg:

```
int B = 10;
#pragma omp parallel private(B)
    {
        B=...;
    }
```

- A un-initialised copy of B is created before the parallel region begins.

FIRSTPRIVATE

- Firstprivate: A private initialized copy of B is created on each thread's stack before the parallel region begins

  - Syntax:

    firstprivate (list)

  - Example:

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

SHARED

- SHARED Clause
  - A shared variable exists in only one memory location and all threads can read or write to that address
  - Syntax:

    shared (list)

- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

2. Runtime Libraries

- Execution environment routines that can be used to control and to query the parallel execution environment

Example routines:

- 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);

- 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()
```

3. Environment Variables

- OMP_SCHEDULE
  - export OMP_SCHEDULE “static"
  - export OMP_SCHEDULE "dynamic"

- OMP_NUM_THREADS
  - export OMP_NUM_THREADS=8

Data Scoping Attribute Clauses

- Purpose : The OpenMP Data Scope attribute clauses are used to explicitly define how variables should be scoped. They include:
- Shared
- PRIVATE
- DEFAULT
- FIRSTPRIVATE
- LASTPRIVATE
- COPYPRIVATE
- COPYIN
- REDUCTION

Shared and Private clauses

- Shared: There exists one instance of this variable which is shared among all the threads
- Private: Each thread in a team of threads has its own local copy of the private variable.
- Syntax: shared(list)

  private(list)

Implicit Rules

```
int i=0;
int n=10;
int a=7;
#pragma omp parallel for
For(i=0;i<n;i++)
{
        int b=a+i;          //Here b is private variable and assigns its local copy to each thread
}
```

- i, n, a and b are 4 variables
- n, a are shared variables
- Loop iteration variables are private by default. So i is private
- Variables which are declared locally within the parallel region are private. Thus b is private

```
int n=10;                  //shared
int a=7;                   //shared
#pragma omp parallel for
For(int i=0;i<n;i++)  //i private
{
      int b=a+i;          // b private
}
```

- It is better to declare the loop iteration variables inside the parallel region

Explicit Rules
Shared

```
#pragma omp parallel for shared(n,a)
for(int i=0; i<n; i++)
{
      int b=a+i;        //
}
```

- The **shared(list)** clause declares that all variables in list are shared
- Shared variables introduce the overhead coz an instance of a variable is shared multiple threads
- Minimize the number of shared variables

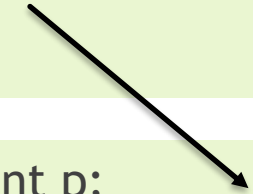
Explicit Rules
Private

```
#pragma omp parallel for shared(n, a) private(b)
for(int i=0; i<n; i++)
{
      int b=a+i;        //b is private
}
```

- The **private(list)** clause declares that all variables in list are private
- OpenMP replicates the private variable and assigns its local copy to each thread.
- Here n, a are shared variables
- And b is private variable

Explicit Rules
Private

```
int p;
#pragma omp parallel private(p)
for(int i=0; i<n; i++)
{
        p=omp_get_thread_num();
}
```



```
int p;
#pragma omp parallel
for(int i=0; i<n; i++)
{
        int p = omp_get_thread_num();
}
```

- We can avoid listing private variables inside the construct inside the parallel region
- Better to declare inside the parallel region

Default
Default(shared)

```
int a, b, c, n;
#pragma omp parallel for default(shared)
for(int i=0; i<n; i++)
{
      …..
}
```

- Observe two uses of Default clause
- Here, a, b, c, n are shared

```
int a, b, c, n;
#pragma omp parallel for default(shared) private(a, b)
for(int i=0; i<n; i++)
{
      // a and b are private variables
      // c and n are shared variables
}
```

- Here, is another use of default(shared)
- Here, c and n are shared
- And, a,b are private

Default
Default(none)

```
int a, b, c, n;
#pragma omp parallel for default(none) shared(b,c ,n)
for(int i=0; i<n; i++)
{
        …..
}
```

- None clauses forces to explicitly specify the data sharing attributes of all variables
- Error as var a data attribute is not specified

```
int a, b, c, n;
#pragma omp parallel for default(none) shared(a, b, c, n)
for(int i=0; i<n; i++)
{
        // a and b are private variables
        // c and n are shared variables
}
```

- Here, var a is included in shared
- Always write parallel regions with the default(none) clause
- Declare private variables inside parallel regions whenever possible

Critical Section

- Critical section prevents multiple threads from accessing a section of code at the same time.
- Only one active thread can update the data referenced by the code.
- SYNTAX:

```
#pragma omp critical [(name)]
{
    code_block
}
```

- If critical sections are not used then manually acquire and release the locks
- What could be an example of critical section
  - Largest number from an Array of 10 numbers

Critical Section

- Critical section prevents multiple threads from accessing a section of code at the same time.

```
#include<omp.h>
int count=0;
Void myFunction()
{
        #pragma omp critical
        count++;
}
```

- What could be an example of critical section
  - Largest number from an Array of 10 numbers

Atomic

- Accessing specific memory location
- Race condition can be avoided
- SYNTAX:

```
#pragma omp atomic update
{
    expression statement;
}
```

FirstPrivate clause

Purpose: FirstPrivate will copy the variable's original value in each thread's private variable

Syntax: firstprivate(var1, var2)

Here, var1 , var2 have instances in each thread.

LastPrivate clause

Purpose: Value from the last loop iteration upon exit from the parallel bloack is assigned to the original variable

Syntax: lastprivate(var1, var2)

Here, var1 , var2 have instances in each thread.

ThreadPrivate clause

Purpose: It makes the variables allocated to the threads persistent across the parallel regions

It also retains the computed values

It preserves the local copy for each thread

Syntax: threadprivate(var1, var2)

Here, var1 , var2 have instances in each thread.

Copyin clause

Purpose: It copies the values from the master thread to all other threads

Its only applicable for threadprivate variables

Syntax: copyin(var1, var2)

Here, var1 , var2 have instances in each thread.

Copyprivate clause

Purpose: The COPYPRIATE clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.

Associated with the SINGLE directive

Syntax: copypriate(var1, var2)

Here, var1 , var2 have instances in each thread.

Conditional If in OpenMP

Purpose: It tells if the code inside the parallel region is executed in parallel or sequential

Syntax: if(expression)

if expression is true, the code inside parallel region executed in parallel else in sequential.

Reduction clause:

Reduction is the accumulation of partial results from each threads into a single final result

A Private copy for each list variables is created for each thread

Syntax: reduction(operator : list)

Operators used with reduction clause:

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

Clause / 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		◆	◆	◆		