Part 3 – Introduction to OpenMP

Addisu G. Semie, PhD
Asst. Prof.,Computational Science Program,
Addis Ababa University
Email: addisugezahegn@gmail.com

Introduction to OpenMP

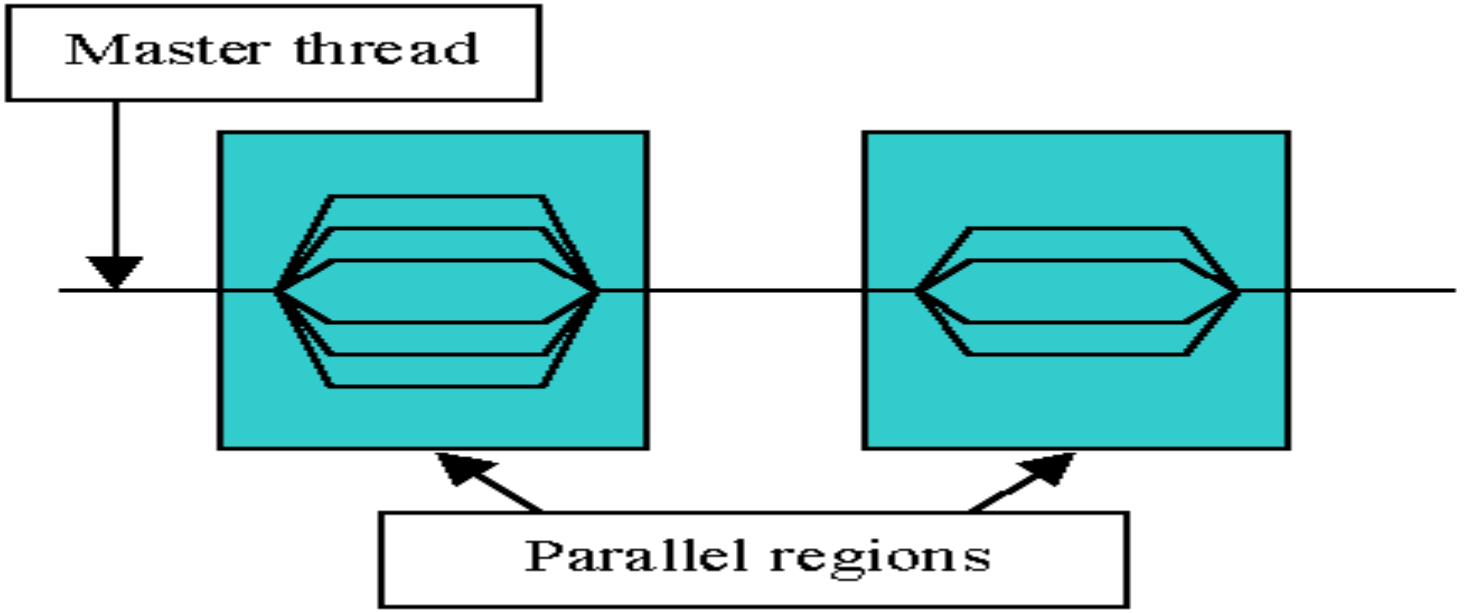
- In this section we will cover:
  - Overview of OpenMP
  - Basic Constructs
  - Parallel Flow Control
  - Synchronization
  - Reduction

Overview of OpenMP

- OpenMP is :
  - an API (Application Programming Interface) that may be used to explicitly direct multi-threaded, shared memory parallelism.
  - NOT a programming language
  - A collection of the **directives, environment variables** and the run-time **library routines**

Overview of OpenMP

- OpenMP:
  - Used for multi-threaded parallel processing
  - Used on shared-memory multi-processor (core) computers
  - Part of program is a single thread and part is multi-threaded



Overview of OpenMP

- OpenMP:
  - OpenMP continues to evolve, with new constructs and features being added over time.
  - Initially, the API specifications were released separately for C and Fortran. Since 2005, they have been released together.
  - Refer http://openmp.org to see the OpenMP API release history

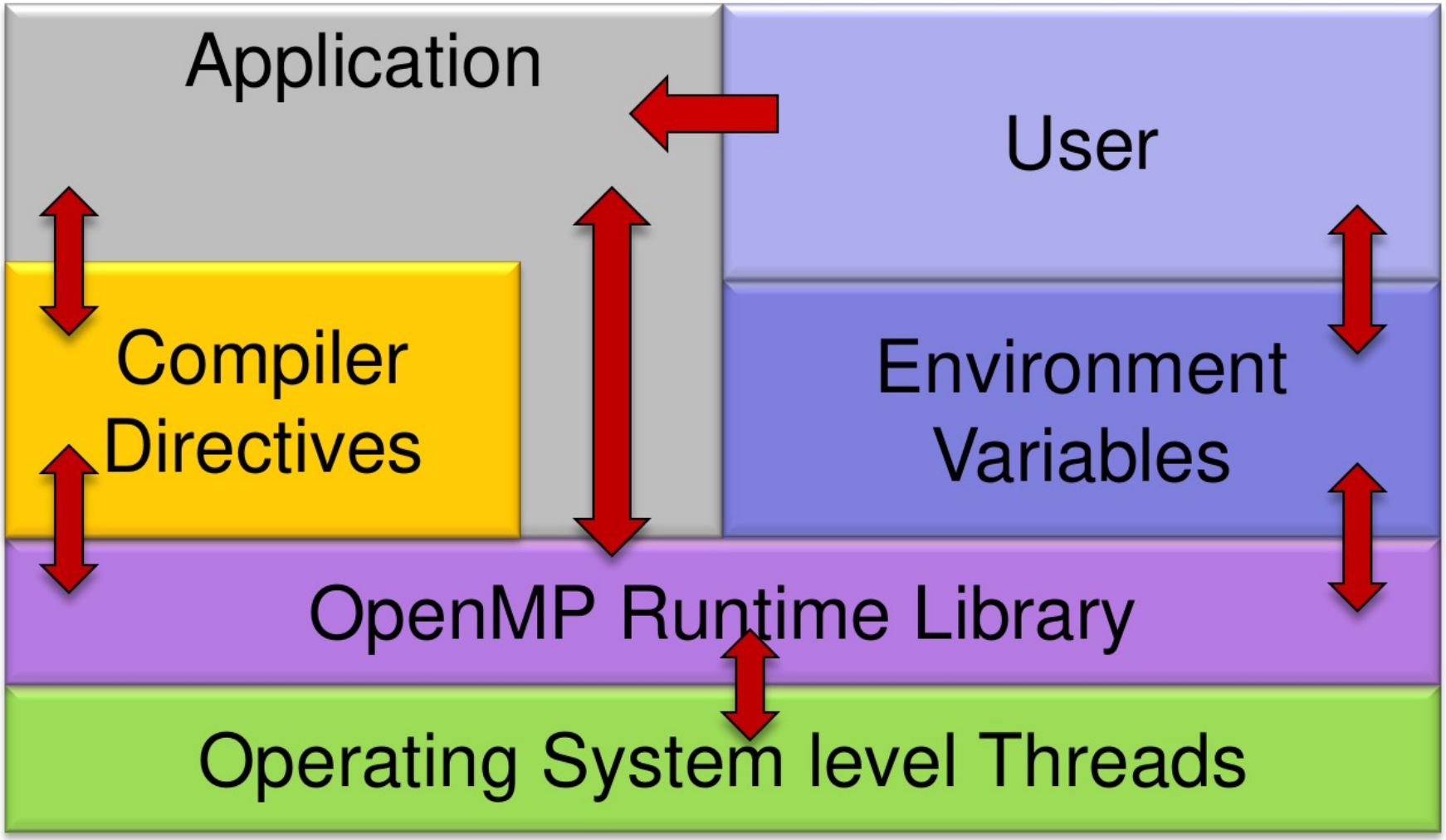
Components of OpenMP

Directives (44)
Parallel regions
Work sharing
Synchronization
Data scope attributes: • private • first private • last private • shared • reduction
Orphaning

Runtime library routines (35)
Number of threads
Thread ID
Dynamic thread adjustment
Nested Parallelism
Timers
API for locking

Environment variables (13)
Number of threads
Scheduling type
Dynamic thread adjustment
Nested Parallelism

OpenMP Architecture



Inspired by **OpenMp.org** introductory slides

Overview of OpenMP

- Advantage of OpenMP
  - Prevalence of multi-core computers
  - Requires less code modification than using MPI
  - OpenMP directives can be treated as comments if OpenMP is not available
  - Directives can be added incrementally

Overview of OpenMP

- Disadvantage of OpenMP
  - OpenMP codes cannot be run on distributed memory computers (exception is Intel's OpenMP)
  - Requires a compiler that supports OpenMP
  - Limited by the number of processors available on a single computer
  - Often have lower parallel efficiency
    - Rely more on parallelizable loops
    - Tend to have a higher % of serial code

Overview of OpenMP

- Examples of Applications and Libraries That Use OpenMP
  - Applications
    - Matlab
    - Mathematica
  - Libraries
    - Intel Math kernel Library (MKL)
    - AMD Core Math Library (ACML)
    - GNU Scientific Library (GSL)

Overview of OpenMP

- Approaches to Parallelism Using OpenMP Applications
  - Two main approaches:
    - Loop-level
    - Parallel regions

Overview of OpenMP

- Loop-Level Parallelism
  - Sometimes called fine-grained parallelism
  - Individual loops parallelized
  - Each thread assigned a unique range of the loop index
  - Execution starts on a single serial thread
  - Multiple threads are spawned inside a parallel loop
  - After parallel loop execution is serial
  - Relatively easy to implement

```
!     Fortran example

!$omp parallel do
   do i = 1, n
      a(i) = b(i) + c(i)
   enddo
```

```
/* C/C++ Example */

  #pragma omp parallel for
   for(i=1; i<=n; i++)
      a[i] = b[i] + c[i]
```

Overview of OpenMP

- Parallel Regions Parallelism
  - Sometimes called coarse-grained parallelism
  - Any sections of codes can be parallelized (not just loops)
  - Using the thread identifier to distribute the work
  - Execution starts on a single serial thread
  - Multiple threads are started for parallel regions (not necessarily at a loop)
  - Ends on a single serial thread

Overview of OpenMP

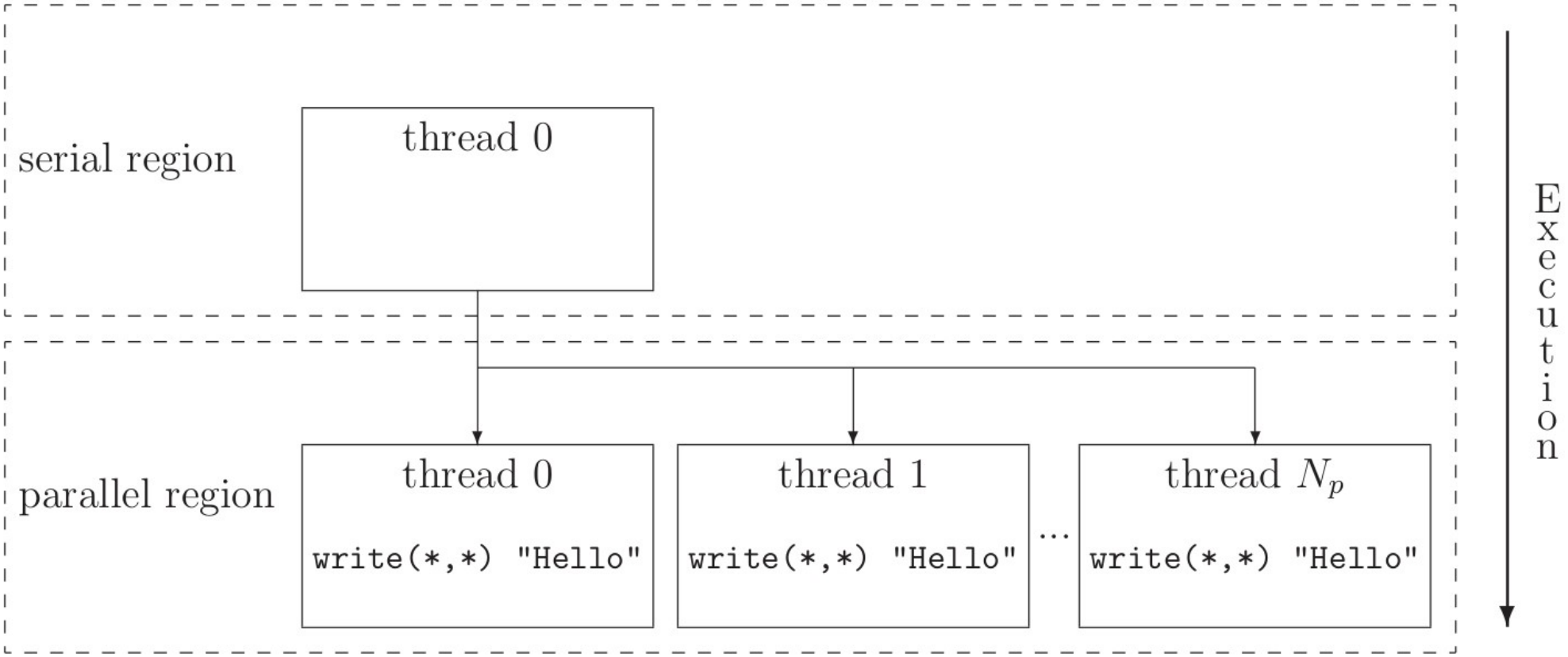
- OpenMP:
  - Parallel regions need to be marked
    - Works by adding compiler directives to the code (This is invisible to non-openMP compilers)
    - *!$OMP* – the OpenMP- compliant compiler knows that the following information in the line is an OpenMP directive – defines parallel regions

```
!$OMP PARALLEL
      write(*,*) "Hello"
!$OMP END PARALLEL
```

    - !$ OMP END PARALLEL has an **implied synchronization**

Overview of OpenMP

- OpenMP:
  - Working principle of the *!$OMP PARALLEL/!$OMP END PARALLEL directive-pair*



Overview of OpenMP

- Basic OpenMP functions
  - omp_get_thread_num() - get the thread rank in a parallel region
  - omp_set_num_threads(nthreads) - affects the number of threads to be used for subsequent parallel regions that do not specify a **num_threads** clause.
  - omp_get _num_threads() - get the number of threads used in a parallel region

Overview of OpenMP

- OpenMP is a portable, threaded, shared-memory programming specification with “light” syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into serial regions and parallel regions, rather than concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

Basic Constructs

Basic Constructs

- Constructs allow to distribute a given task over the different threads
- To invoke library routine, put the following in top of your code in:
  - FORTRAN ‘USE OMP_LIB’
  - C/C++ ‘#include <omp.h>’

OpenMP Execution Model:

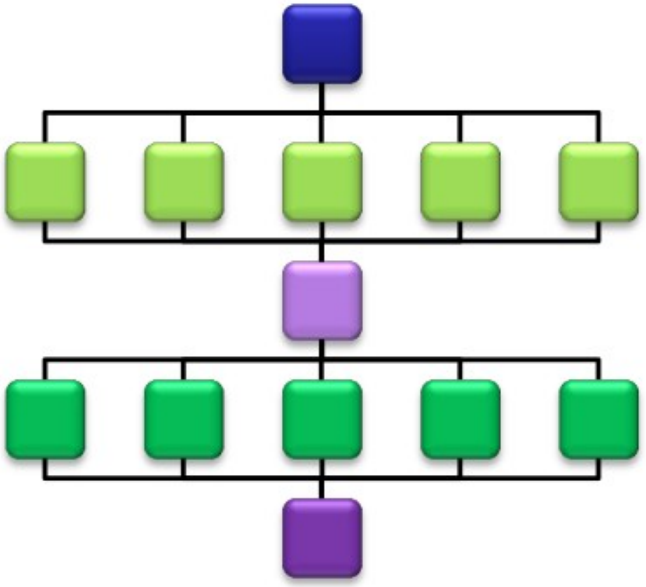
Sequential Part (master thread)

Parallel Region (group of threads)

Sequential Part (master thread)

Parallel Region (group of threads)

Sequential Part (master thread)



Basic Constructs

- HelloWorld in OpenMP

```
#include <omp.h>

main ()
{
   int nthreads, tid;
   #pragma omp parallel private(nthreads, tid)
   {
      tid = omp_get_thread_num();
      printf("Hello World from thread = %d\n", tid);
      if (tid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }
   }
}
```

OpenMP directive to indicate START segment to be parallelized

Code segment that will be executed in parallel

OpenMP directive to indicate END segment to be parallelized

Basic Constructs

- HelloWorld in OpenMP in FORTRAN

```
program hello
USE OMP_LIB
IMPLICIT NONE
INTEGER :: tid, t_total
INTEGER, PARAMETER :: nthreads = 12
        call omp_set_num_threads(nthreads)
        !Do this part in parallel
        !$omp parallel private(tid, t_total)
                tid = omp_get_thread_num()
                print *, "Hello World from threads = ", tid
                if(tid == 0) then
                        t_total = omp_get_num_threads()
                        print*, "Number of threads =", t_total
                end if
        !$omp end parallel

end program hello
```

Basic Constructs

- Compiling OpenMP programs

C :

- Case sensitive directives
- Syntax :
  - #pragma omp directive [clause [clause]..]
- Compiling OpenMP source code :
  - **(IBM xlc compiler) : xlc_r -q64 -O2 -qsmp=omp file_name.c –o exe_file_name**
  - **(Linux C compiler) : gcc –o exe_file_name –openmp file_name.c**

Fortran :

- Case insensitive directives
- Syntax :
  - !$OMP directive [clause[[,] clause]…] (free format)
  - !OMP / COMP / *$OMP directive [clause[[,] clause]…] (free format)
- Compiling OpenMP source code :
  - **(IBM xlf compiler) : xlf_r -q64 -O2 -qsmp=omp file_name.f –o exe_file_name**
  - **(Linux Fortran compiler) : gfort -o exe_file_name –openmp file_name.f**

Basic Constructs

- How to Compile an OpenMP Program?

Compiler	Compiler Options	Default behavior for # of threads (OMP_NUM_THREADS not set)
GNU (gcc, g++, gfortran)	-fopenmp	as many threads as available cores
Intel (icc ifort)	-openmp	as many threads as available cores
Portland Group (pgcc,pgCC,pgf77,pgf90)	-mp	one thread

Environment Variable **OMP_NUM_THREADS** sets the number of threads

Basic Constructs

- Environment variables

Environment Variable:	OMP_NUM_THREADS
Usage : bash/sh/ksh: csh/tcsh	OMP_NUM_THREADS ***n*** *export* OMP_NUM_THREADS=8 *setenv* OMP_NUM_THREADS=8
Description:	
Sets the number of threads to be used by the OpenMP program during execution.	

Environment Variable:	OMP_DYNAMIC
Usage : bash/sh/ksh: csh/tcsh	OMP_DYNAMIC {TRUE|FALSE} *export* OMP_DYNAMIC=TRUE *setenv* OMP_DYNAMIC="TRUE"
Description:	
When this environment variable is set to TRUE the maximum number of threads available for use by the OpenMP program is $OMP_NUM_THREADS.	

Basic Constructs

- Environment variables

Environment Variable:	OMP_SCHEDULE
Usage : bash/sh/ksh: csh/tcsh	OMP_SCHEDULE ***"schedule,[chunk]"*** *export* OMP_SCHEDULE static,N/P *setenv* OMP_SCHEDULE="GUIDED,4"

Description:

Only applies to for and parallel for directives. This environment variable sets the schedule type and chunk size for all such loops. The chunk size can be provided as an integer number, the default being 1.

Environment Variable:	OMP_NESTED
Usage : bash/sh/ksh: csh/tcsh	OMP_NESTED {TRUE|FALSE} *export* OMP_NESTED FALSE *setenv* OMP_NESTED="FALSE"

Description:

Setting this environment variable to TRUE enables multi-threaded execution of inner parallel regions in nested parallel regions.

Basic Constructs

- Execution of an OpenMP code :
  - On encountering the C construct **#pragma omp** parallel or the Fortran equivalent **!$omp parallel**, n extra threads are created
  - **omp_get_num_threads()** returns the number of execution threads (**nthreads**) that can be utilized.
  - Code after the parallel directive is executed independently on each of the nthreads.
  - On encountering the C construct **}** or Fortran equivalent **!$omp end parallel**, indicates the end of parallel execution of the code segment, the n extra threads are deactivated and normal sequential execution begins.

Basic Constructs

- Hello World

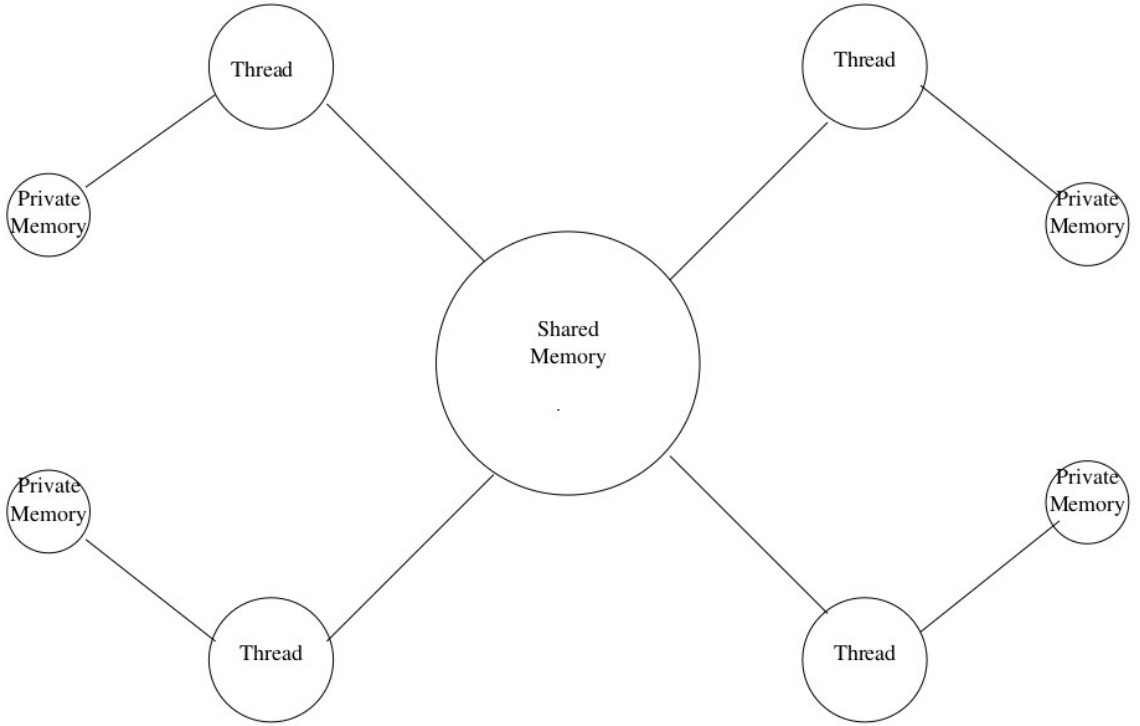
```
(base) addisu@adew:/data/Scripts/cdsc604/openmp$ vi hello_mp.f90
(base) addisu@adew:/data/Scripts/cdsc604/openmp$ gfortran -fopenmp hello_mp.f90 -o hellof
(base) addisu@adew:/data/Scripts/cdsc604/openmp$ export OMP_NUM_THREADS=8
(base) addisu@adew:/data/Scripts/cdsc604/openmp$ ./hellof
 Hello World from threads =           0
 Hello World from threads =           2
 Hello World from threads =           6
 Hello World from threads =           4
 Hello World from threads =           3
 Hello World from threads =           1
 Hello World from threads =           5
 Hello World from threads =           7
(base) addisu@adew:/data/Scripts/cdsc604/openmp$
```

Data Environment

- Data Environment:
  - OpenMP program always begins with a single thread of control master thread
  - Context associated with the master thread is also known as Data Environment
  - Context of the master thread remains valid throughout the execution of the program

Data Environment

- Data Environment:
  - The OpenMP parallel construct may be used to either **share a single copy** of the context with all the threads or provide each of the threads with a **private copy** of the context.



Each thread has access to the shared memory as well as its own private memory that none of the other threads can see or touch

Shared Data

- The default context of a variable is determined by the following rules:
  - Variables with static storage duration are shared.
  - Dynamically allocated objects are shared.
  - Variables with automatic storage duration that are declared in a parallel region are private.
  - All variables defined outside a parallel construct become shared when the parallel region is encountered.
  - Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
  - Memory allocated within a parallel loop by the allocation function persists only for the duration of one iteration of that loop, and is private for each thread.

The following code segments show examples of these default rules:

```
int E1;                                /* shared static      */

void main (argvc,...) {                /* argvc is shared    */
   int i;                               /* shared automatic   */

void *p = malloc(...);                /* memory allocated by malloc    */
                                       /* is accessible by all threads */
                                       /* and cannot be privatized      */

#pragma omp parallel firstprivate (p)
   {
     int b;                            /* private automatic   */
     static int s;                     /* shared static       */

     #pragma omp for
     for (i =0;...) {
       b = 1;                          /* b is still private here !     */
       foo (i);                        /* i is private here because it */
                                       /* is an iteration variable      */
      }


#pragma omp parallel
     {
       b = 1;                          /* b is shared here because it   */
                                       /* is another parallel region    */
     }
   }
 }


int E2;                                /*shared static */

void foo (int x) {                     /* x is private for the parallel */
                                        /* region it was called from     */

int c;                                /* the same */
 ... }
```

Shared Data

Some OpenMP clauses enable you to specify visibility context for selected data variables. A brief summary of data scope attribute clauses are listed below:

Data scope attribute clause	Description
private	The **private** clause declares the variables in the list to be private to each thread in a team.
firstprivate	The **firstprivate** clause provides a superset of the functionality provided by the private clause. The private variable is initialized by the original value of the variable when the parallel construct is encountered.
lastprivate	The **lastprivate** clause provides a superset of the functionality provided by the private clause. The private variable is updated after the end of the parallel construct.
shared	The **shared** clause declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The **reduction** clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
default	The **default** clause allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct.

Shared Data

- Data Environment:
  - OpenMP data scoping clauses allows programmer to decide whether a variable's execution context i.e. should the variable be shared or private.
  - **Shared** :
    - A variable will have a single storage location in memory for the duration of the parallel construct, i.e. references to a variable by different threads access the same memory location.
    - That part of the memory is shared among the threads involved, hence modifications to the variable can be made using simple read/write operations
    - Modifications to the variable by different threads is managed by underlying shared memory mechanisms

Shared Data

- Data Environment:
  - Private :
    - A variable will have a separate storage location in memory for each of the threads involved for the duration of the parallel construct.
    - All read/write operations by the thread will affect the thread’s private copy of the variable .
  - Reduction :
    - Exhibit both shared and private storage behavior. Usually used on objects that are the target of arithmetic reduction.
    - Example : summation of local variables at the end of a parallel construct

Parallel Flow Control

Parallel Flow Control

- Work-Sharing Directives
  - Divide the execution of the enclosed block of code among the group of threads.
  - They do not launch new threads.
  - No implied barrier on entry
  - Implicit barrier at the end of work-sharing construct
  - Commonly used Work Sharing constructs :
    - for directive (C/C++) or DO construct (Fortran) shares iterations of a loop across a group of threads
    - sections directive : breaks work into separate sections between the group of threads; such that each thread independently executes a section of the work.
    - single directive: serializes a section of code

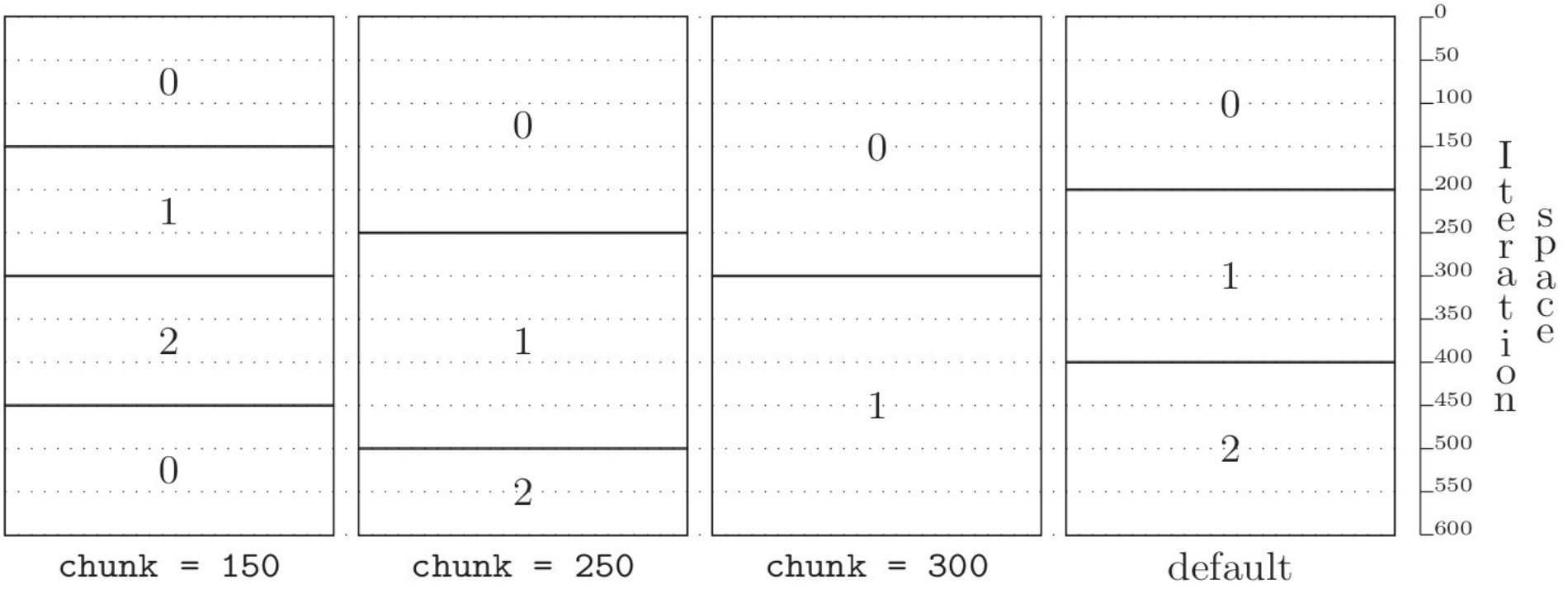
Parallel Flow Control

- Schedule clause
  - A specification of how iterations of associated loops are divided into contiguous non-empty subsets
    - We call each of the contiguous non-empty subsets a **chunk**
  - The size of a chunk, denoted as **chunk_size** must be a positive integer.
  - The SCHEDULE clause accepts two parameters, !$OMP DO SCHEDULE(type, chunk)
    - type - specifies the way in which the work is distributed over the threads. **Four** different options of scheduling exist
    - chunk - is an optional parameter specifying the size of the work given to each thread

Parallel Flow Control

- Schedule clause
  - STATIC : iterations are divided into pieces of size chunk and are statically assigned to each of the threads in a round robin fashion

```
!$OMP DO SCHEDULE(STATIC, chunk)
do i = 1, 600
...
enddo
```



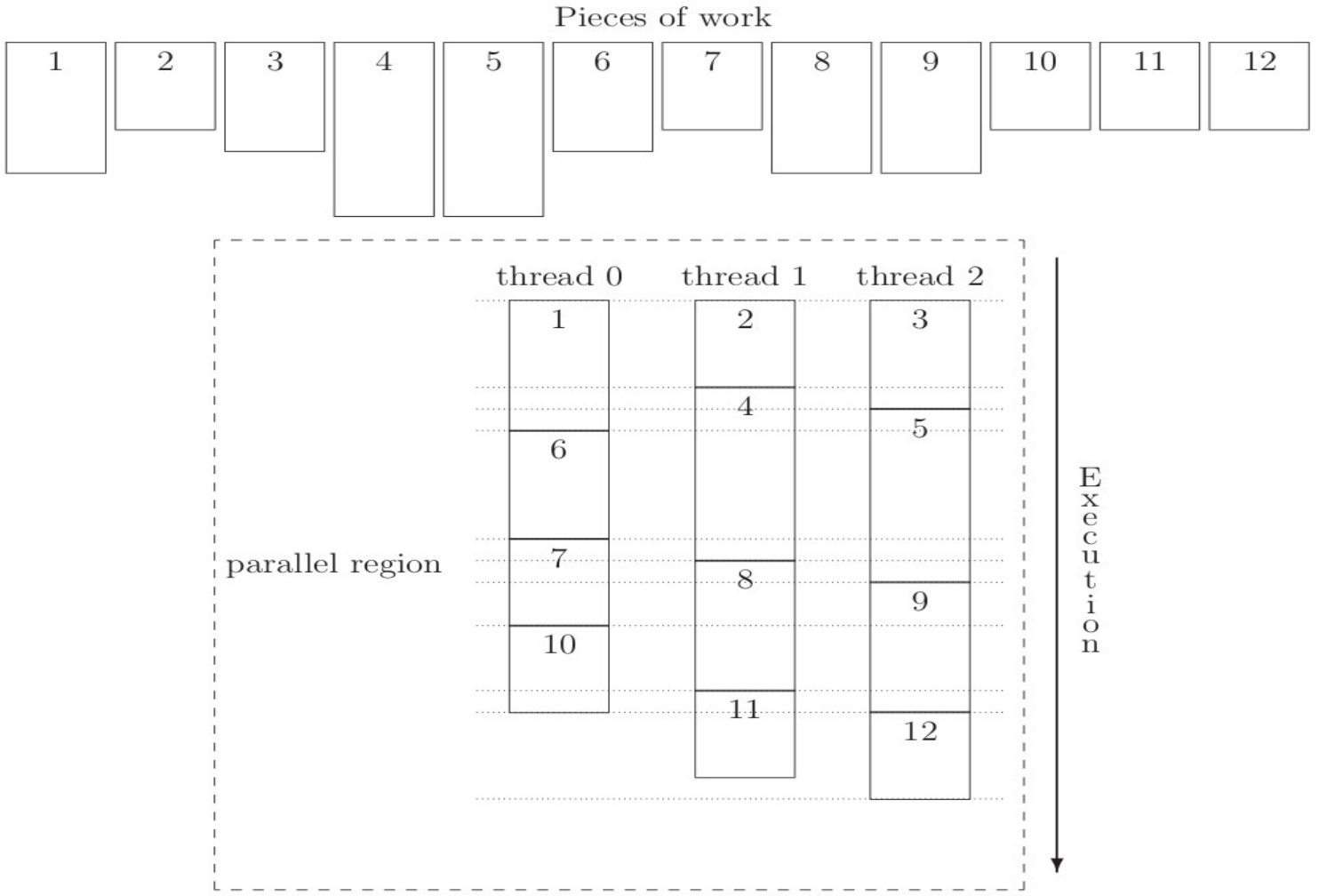
Parallel Flow Control

- Schedule clause
  - DYNAMIC : iterations divided into pieces of size chunk and dynamically assigned to a group of threads.
  - After a thread finishes processing a chunk, it is dynamically assigned the next set of iterations.
  - If the optional parameter (chunk) is not given, then a size equal to one iteration is considered.

```
!$OMP DO SCHEDULE(DYNAMIC, chunk)
do i = 1, 600
...
enddo
!$OMP END DO
```

Parallel Flow Control

- Schedule clause
  - SCHEDULE(DYNAMIC,chunk) clause



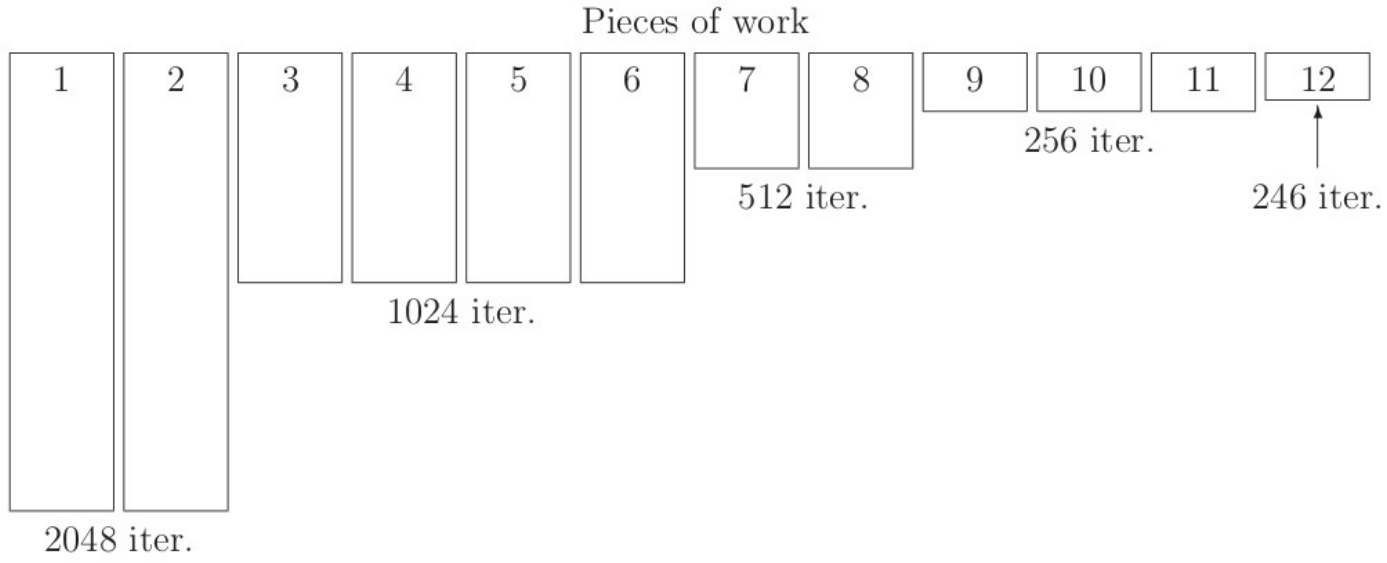
Parallel Flow Control

- Schedule clause
  - **GUIDED** : For a chunk of size 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1.
  - For a chunk with value k, the same algorithm is used in determining the chunk size with the constraint that no chunk should have less than k chunks except the last chunk.

```
!$OMP DO SCHEDULE(GUIDED, 256)
do i = 1, 10230
...
enddo
!$OMP END DO
```

Parallel Flow Control

- Schedule clause
  - SCHEDULE(GUIDED,chunk) clause on a do-loop with 10230 iterations and with chunk = 256.



  - Once the pieces of work have been created, the way in which they are distributed among the available threads is the same as in the DYNAMIC scheduling method.

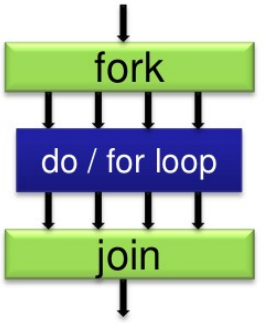
Parallel Flow Control

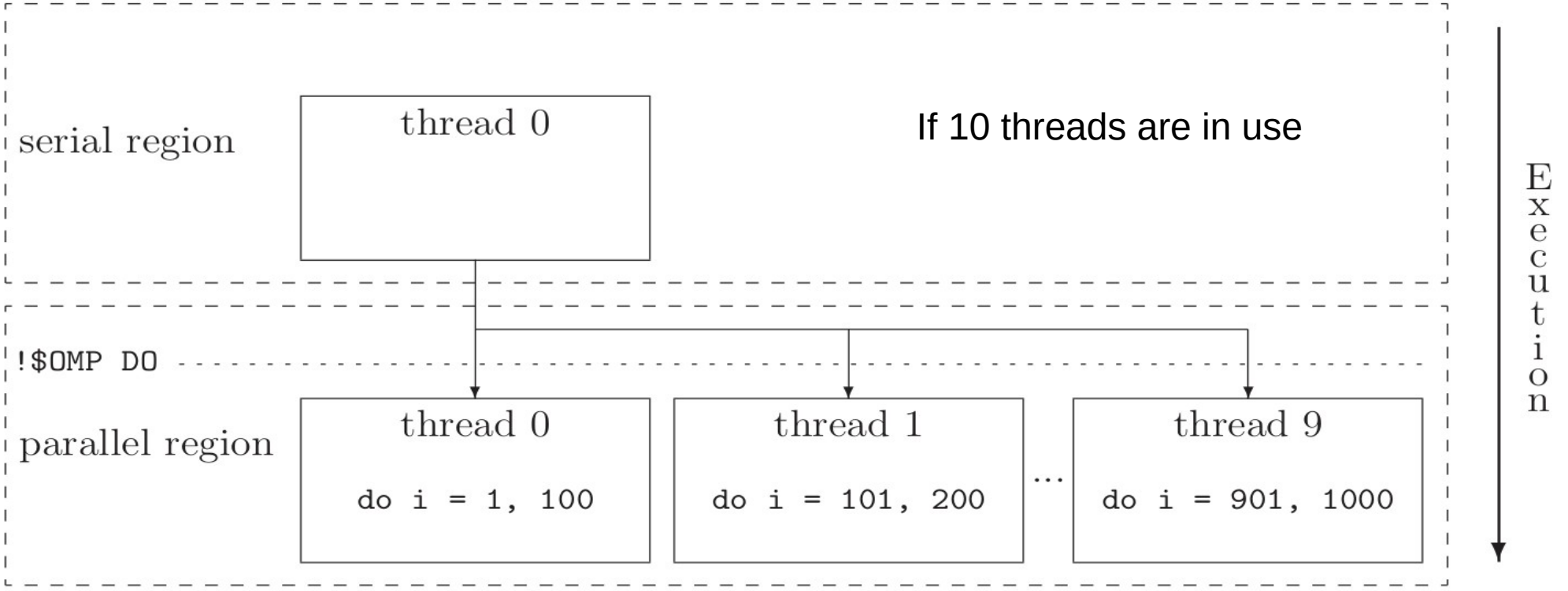
- Schedule clause
  - RUNTIME - enable to modify the way the work is distributed among the threads during runtime.
    - !$OMP DO SCHEDULE(RUNTIME)
  - In this case, the content of the environment variable OMP SCHEDULE specifies the scheduling method to be used.

Parallel Flow Control

- !$OMP DO/!$OMP END DO - distributes the do-loop over the different threads: each thread computes part of the iterations.

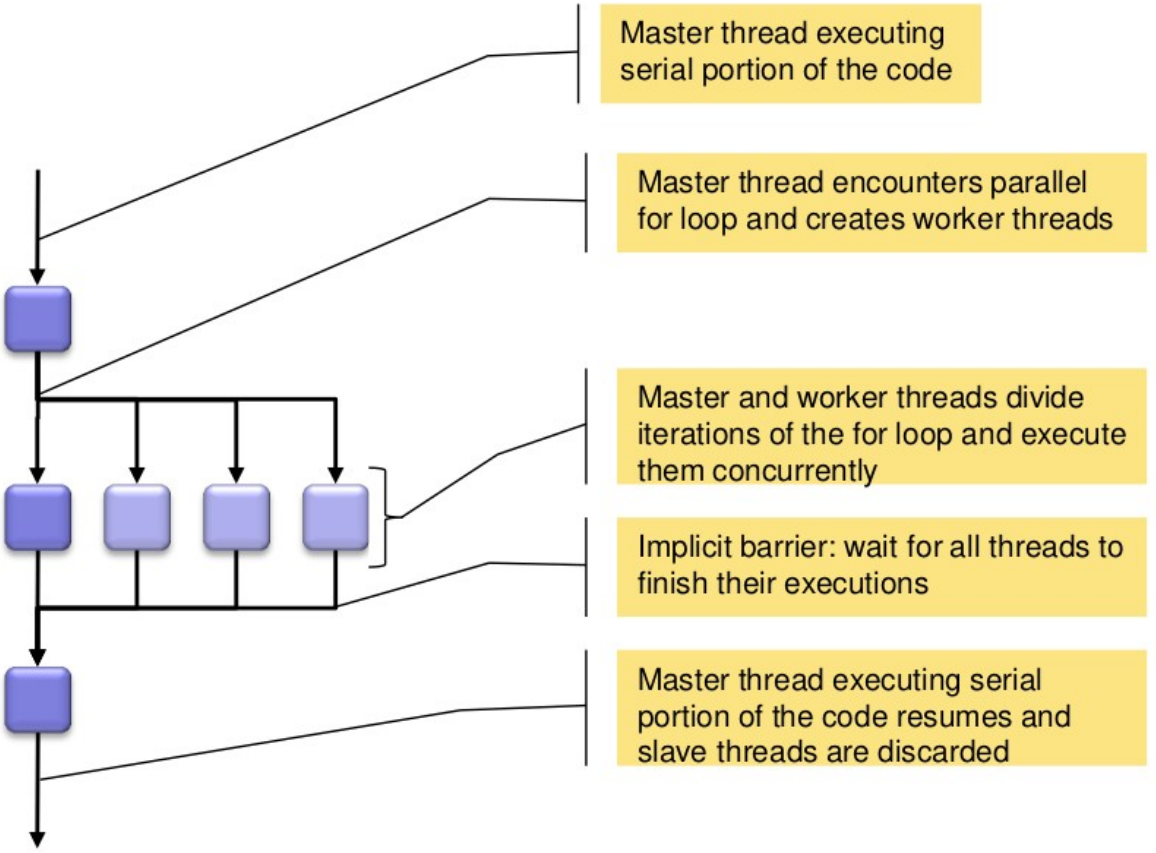
```
!$OMP DO
      do i = 1, 1000
          ...
      end do
!$OMP END DO
```





Parallel Flow Control

- If **nowait** is specified then the threads do not wait for synchronization at the end of a parallel loop
- The **schedule** clause describes how iterations of a loop are divided among the threads in the team



Parallel Flow Control

- OpenMP work sharing Constructs

```
#include <omp.h>
#define N     16
main ()
{
int i, chunk;
float a[N], b[N], c[N];
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = 4;
printf("a[i]  +  b[i]  =  c[i] \n");
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section */
for (i=0; i < N; i++)
    printf(" %f   +   %f   =   %f  \n",a[i],b[i],c[i]);
}
```

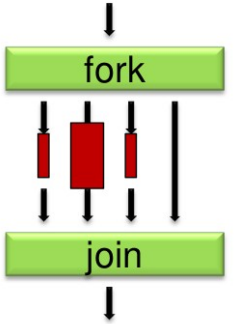
Initializing the vectors a[i], b[i]

Instructing the runtime environment that a,b,c,chunk are shared variables and I is a private variable

The nowait ensures that the child threads donot synchronize once their work is completed

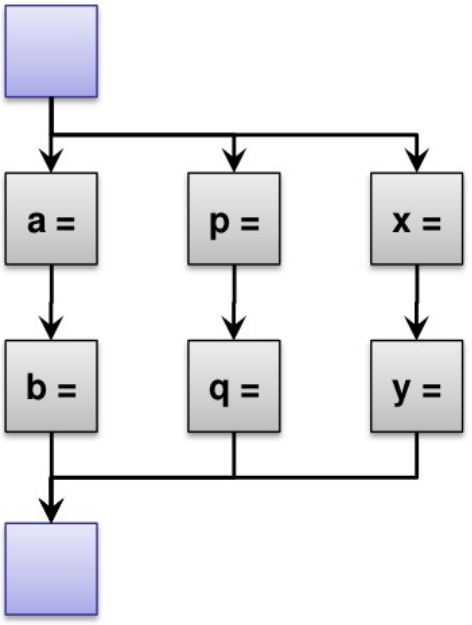
Load balancing the threads using a DYNAMIC policy where array is divided into chunks of 4 and assigned to the threads

Parallel Flow Control



- OpenMP sections directive:
  - Is a non iterative work sharing construct.
  - Independent section of code are nested within a sections directive
  - It specifies enclosed section of codes between different threads
  - Code enclosed within a section directive is executed once by a thread in the team

```
#pragma omp parallel private(p)
{
#pragma omp sections
{{  a=…;
   b=…;}
 #pragma omp section
 {  p=…;
   q=…;}
 #pragma omp section
 {  x=…;
   y=…;}
 } /* omp end sections */
} /* omp end parallel */
```



Parallel Flow Control

- OpenMP Sections

```
#include <omp.h>
#define N  16
main (){
int i;
float a[N], b[N], c[N], d[N];
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.5;
#pragma omp parallel shared(a,b,c,d) private(i)
 {
 #pragma omp sections nowait
  {
  #pragma omp section
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  #pragma omp section
  for (i=0; i < N; i++)
    d[i] = a[i] * b[i];
  } /* end of sections */
 } /* end of parallel section */
```

Sections construct that encloses the section calls

Section : that computes the sum of the 2 vectors

Section : that computes the product of the 2 vectors

Parallel Flow Control

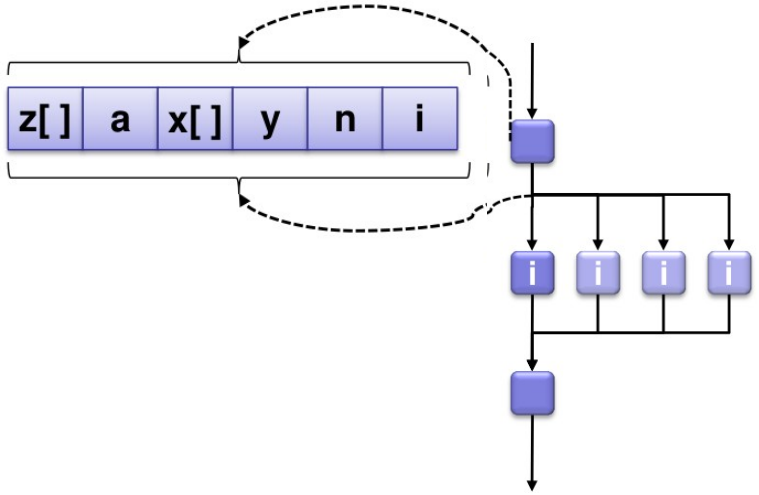
- Understanding variables in OpenMP
  - Shared variable z is modified by multiple threads

```
#pragma omp parallel for
for (i=0; i<n; i++)
    z[i] = a*x[i]+y
```

  - Each iteration reads the scalar variables a and y and the array element x[i]
  - a,y,x can be read concurrently as their values remain unchanged.

Parallel Flow Control

- Understanding variables in OpenMP
  - Each iteration writes to a distinct element of z[i] over the index range. Hence write operations can be carried out concurrently with each iteration writing to a distinct array index and memory location
  - The parallel for directive in OpenMP ensure that the for loop index value (i in this case) is private to each thread.



Synchronization

Synchronization

- “Communication” mainly through read write operations on shared variables
- **Synchronization** defines the mechanisms that help in coordinating execution of multiple threads (that use a shared context) in a parallel program.
- Without synchronization, multiple threads accessing shared memory location may cause conflicts by :
  - Simultaneously attempting to modify the same location
  - One thread attempting to read a memory location while another thread is updating the same location.

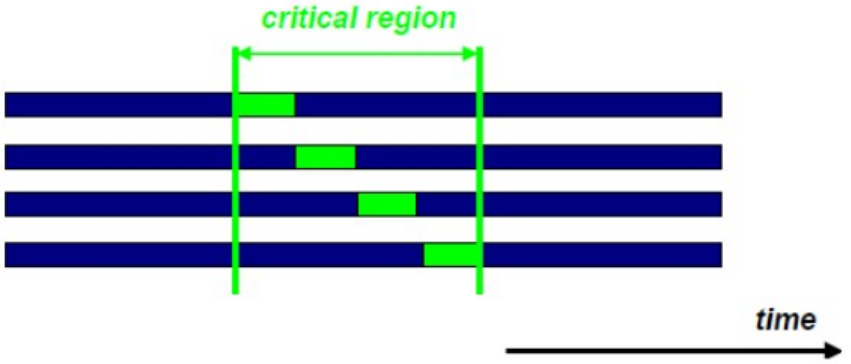
Synchronization

- Threads communicate through shared variables. Uncoordinated access of these variables can lead to undesired effects.
  - E.g. two threads update (write) a shared variable in the same step of execution, the result is dependent on the way this variable is accessed. This is called a **race condition**.
- To prevent race condition, the access to shared variables must be synchronized.
- Synchronization can be time consuming.

Synchronization: critical

- Mutual exclusion: only one thread at a time can enter a critical region.

```
{
   double res;
    #pragma omp parallel
    {
        double B;
         int  i, id, nthrds;
         id = omp_get_thread_num();
         nthrds = omp_get_num_threads();
          for(i=id; i<niters; i+=nthrds){
              B = some_work(i);
               #pragma omp critical
               consume(B,res);
          }
    }
}
```



Threads wait here: only one thread at a time calls consume(). So this is a piece of sequential code inside the for loop.

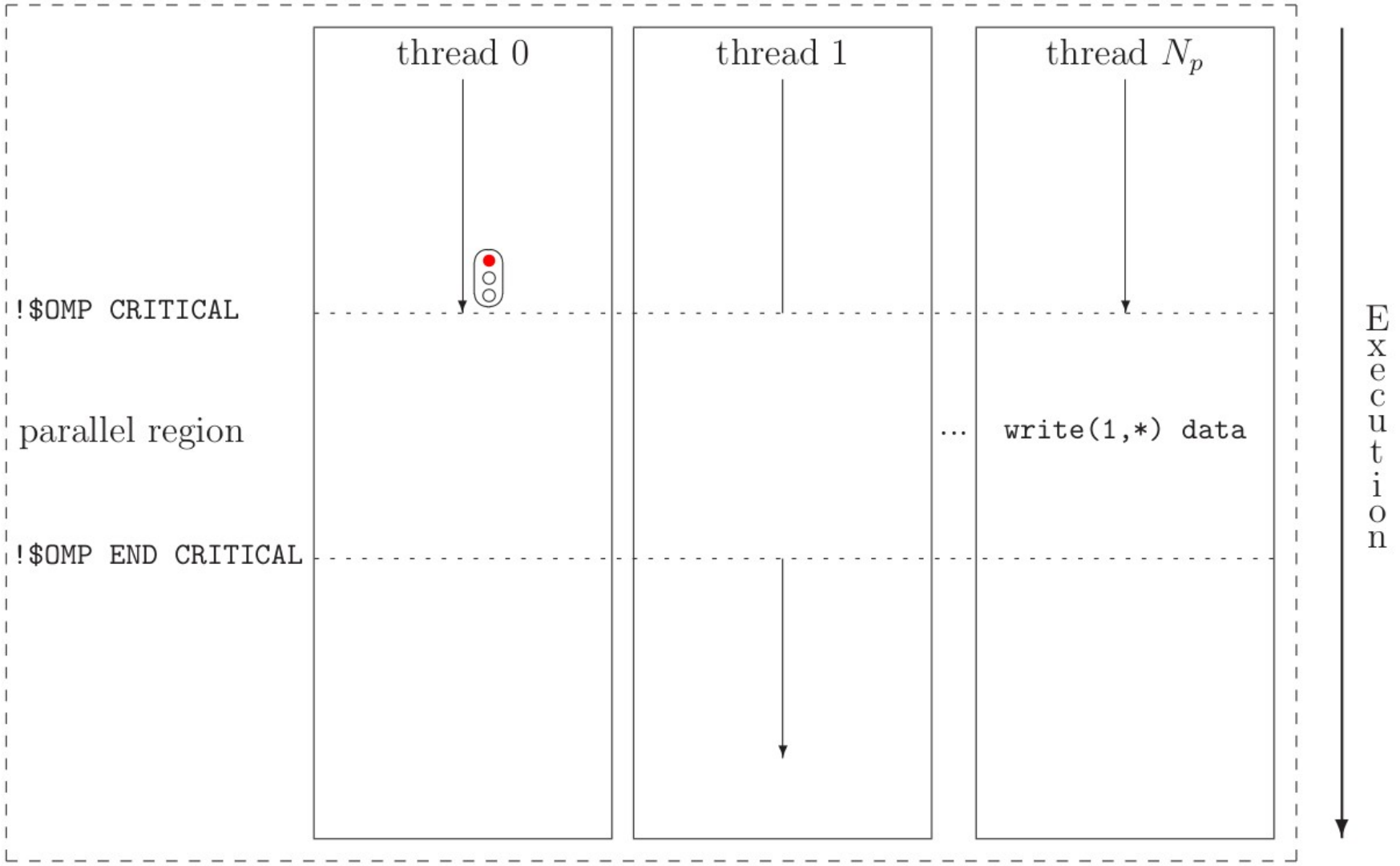
Synchronization: critical

- Mutual exclusion: only one thread at a time can enter a critical region.

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(TID,sumLocal)
  {
     TID = omp_get_thread_num();
     sumLocal = 0;
     #pragma omp for
       for (i=0; i<n; i++)
          sumLocal += a[i];
     #pragma omp critical (update_sum)
      {
       sum += sumLocal;
       printf("TID=%d: sumLocal=%d sum = %d\n",TID,sumLocal,sum);
      }
  } /*-- End of parallel region --*/
```

Synchronization: critical

- Mutual exclusion: only one thread at a time can enter a critical region.



Synchronization: atomic

- Atomic provides mutual exclusion but only applies to the load/update of a memory location.
- This is a lightweight, special form of a critical section.
- It is applied only to the (single) assignment statement that immediately follows it.

```
{
   …
    #pragma omp parallel
    {
            double tmp, B;
            ….
             #pragma omp atomic
             {
                X+=tmp;
             }
    }
}
```

Atomic only protects the update of X.

Synchronization: atomic

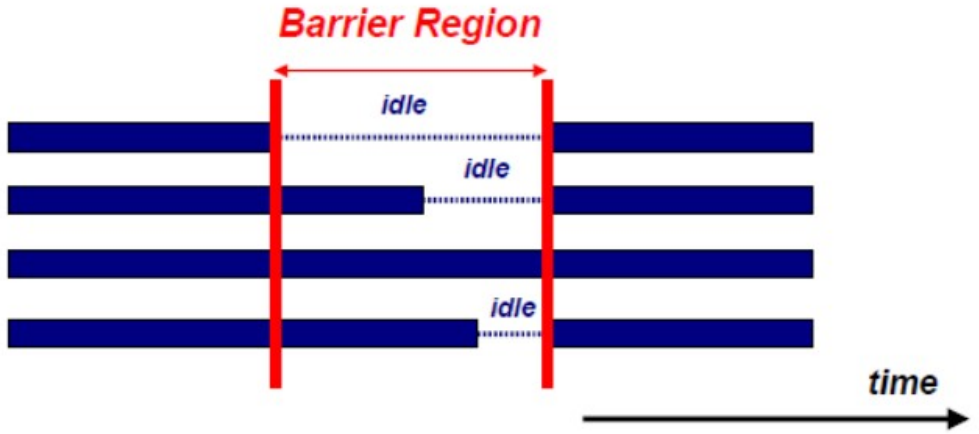
- Atomic provides mutual exclusion but only applies to the load/update of a memory location.

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n,ic) private(i)
   for (i=0; i++, i<n)
    {
      #pragma omp atomic
         ic = ic + 1;
    }
```

“ic” is a counter. The atomic construct ensures that no updates are lost when multiple threads are updating a counter value.

Synchronization: barrier

- Barrier - each threads waits until all threads arrive

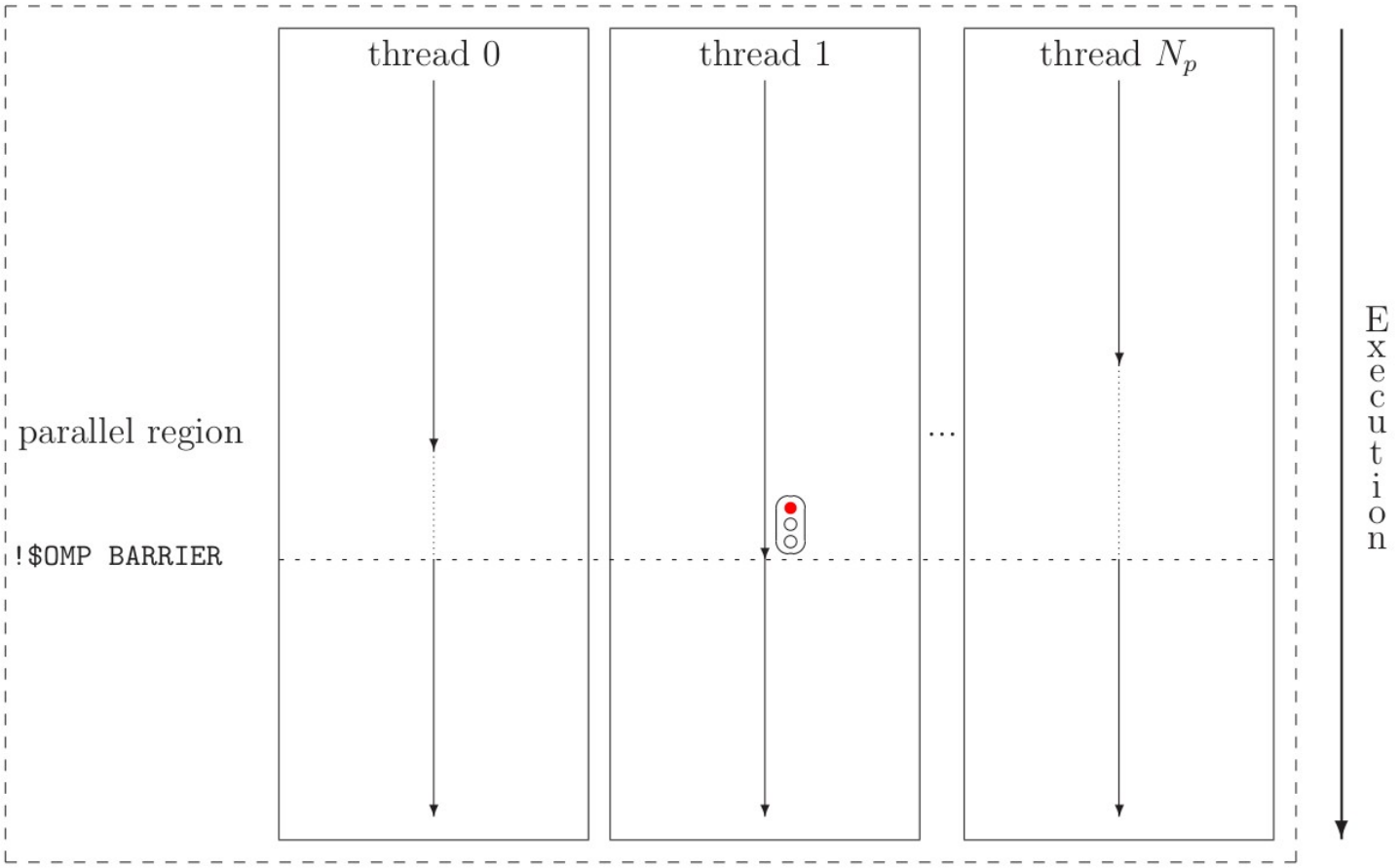


- Barrier is needed, if data is updated asynchronously and data integrity is at risk, examples:
  - Between parts in the code that read and write the same section of memory
  - After one timestep/iteration in a numerical solver
- Barriers are expensive and also may not scale to a large number of processors

Synchronization: barrier

- Barrier - each threads waits until all threads arrive

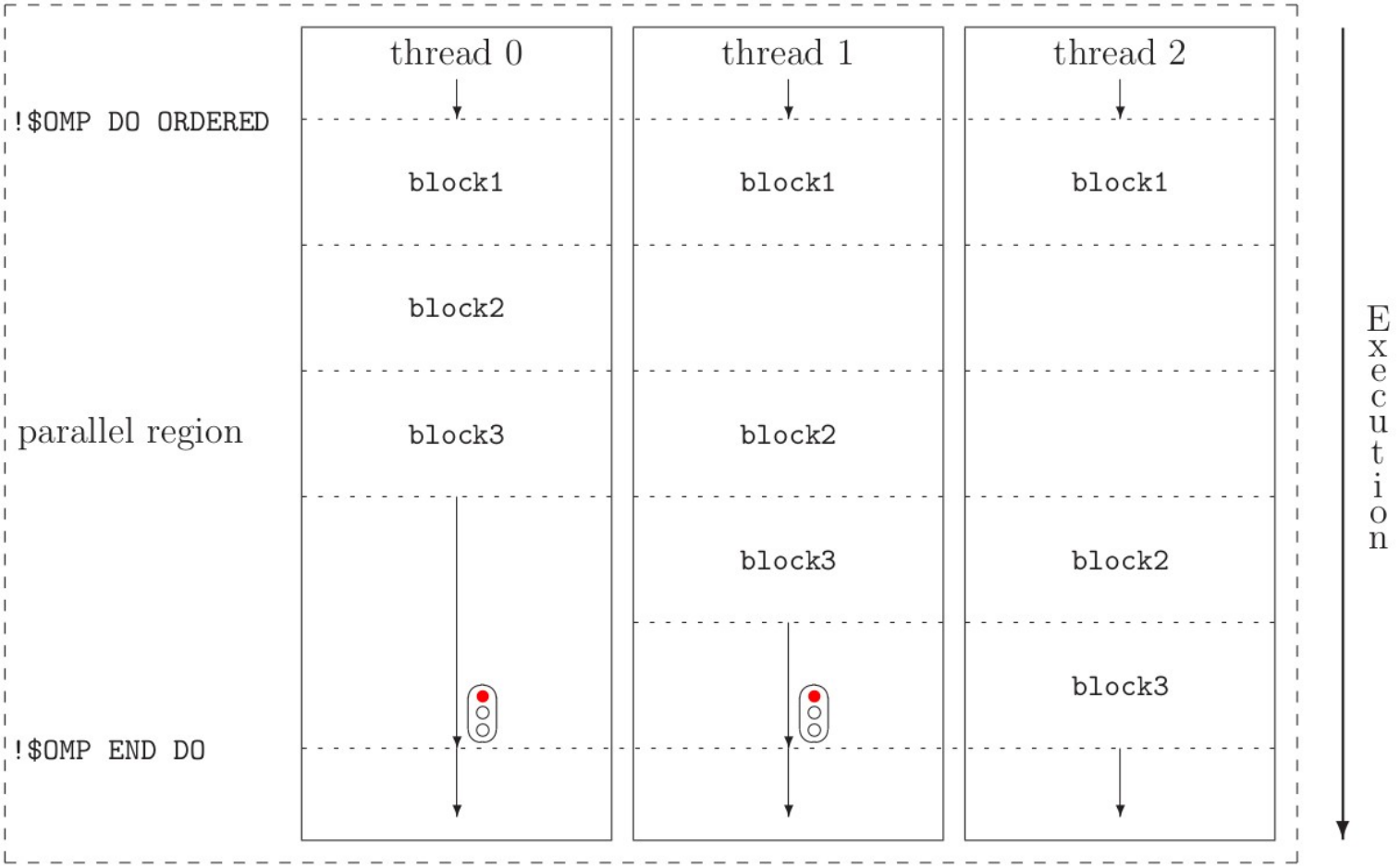
```
!$OMP SECTIONS

 !$OMP SECTION
    !$OMP BARRIER

 !$OMP SECTION
          ...

!$OMP END SECTIONS
```



Synchronization: ordered

- The “ordered” region executes in the sequential order

```
!$OMP DO ORDERED
do i = 1, 100
  block1

  !$OMP ORDERED
    block2
  !$OMP END ORDERED

  block3
enddo
!$OMP END DO
```



Reduction

Reduction

- Performs reduction on **shared variables** in list based on the **operator** provided.
- A private copy of each **variable** in list is created for each thread as if the **PRIVATE** clause had been used
- The resulting private copies are initialized as follows

Operator/Intrinsic	Initialization
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	smallest representable number
MIN	largest representable number
IAND	all bits on
IOR	0
IEOR	0

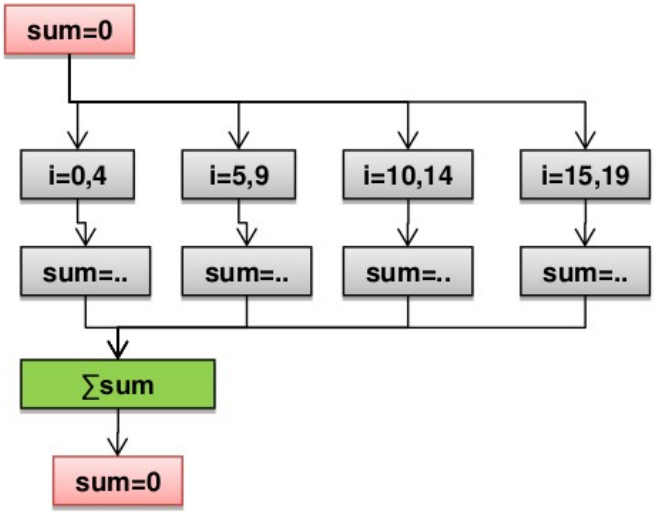
```
operator            = +, *, -, .AND., .OR., .EQV. or .NEQV.

intrincic_procedure = MAX, MIN, IAND, IOR or IEOR
```

x = x operator expr
x = intrinsic_procedure(x, expr_list)

Reduction

- At the end of a reduction, the shared variable contains the result obtained upon combination of the list of variables processed using the operator specified.

```
sum = 0.0
#pragma omp parallel for reduction(+:sum)
  for (i=0; i < 20; i++)
    sum = sum + (a[i] * b[i]);
```



Reduction

```
#include <omp.h>
main ()  {
int   i, n, chunk;
float a[16], b[16], result;
n = 16;
chunk = 4;
result = 0.0;
for (i=0; i < n; i++)
  {
    a[i] = i * 1.0;
    b[i] = i * 2.0;
  }
#pragma omp parallel for default(shared) private(i)  \
    schedule(static,chunk) reduction(+:result)
  for (i=0; i < n; i++)
      result = result + (a[i] * b[i]);
printf("Final result= %f\n",result);
}
```

Reduction example with summation where the result of the reduction operation stores the dotproduct of two vectors

$\sum a[i]*b[i]$

Lab exercises

https://hpc.llnl.gov/tuts/openMP/exercise.html