OPEN MP
SHARED MEMORY DIRECTIVE BASED PROGRAMMING

Dr. Emmanuel S. Pilli

Introduction

- OpenMP is an Application Program Interface (API)
- OpenMP provides a portable, scalable model for developers of shared memory parallel applications.
- The API supports C/C++ and Fortran on a wide variety of architectures.
- This lecture covers most of the major features of OpenMP including its various constructs and directives for specifying parallel regions, work sharing, synchronization and data environment.
- Runtime library functions and environment variables are also covered

What is OpenMP

- An Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory*** parallelism
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- An abbreviation for: **Open Multi-Processing**

What OpenMP is not !

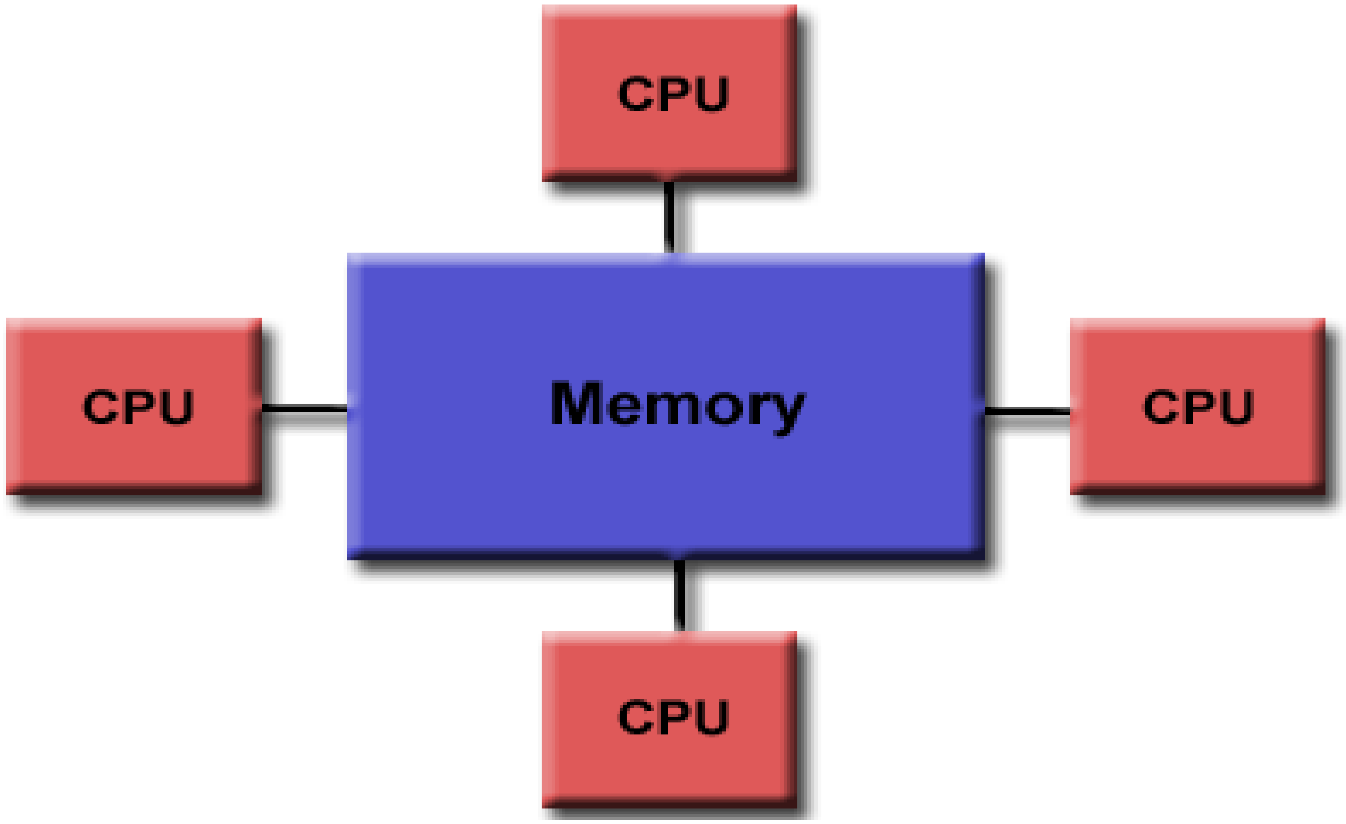
- Meant for distributed memory parallel systems
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, deadlocks, or code sequences that cause a program to be classified as non-conforming
- Designed to handle parallel I/O, but the programmer is responsible for synchronizing input and output.

Goals of OpenMP

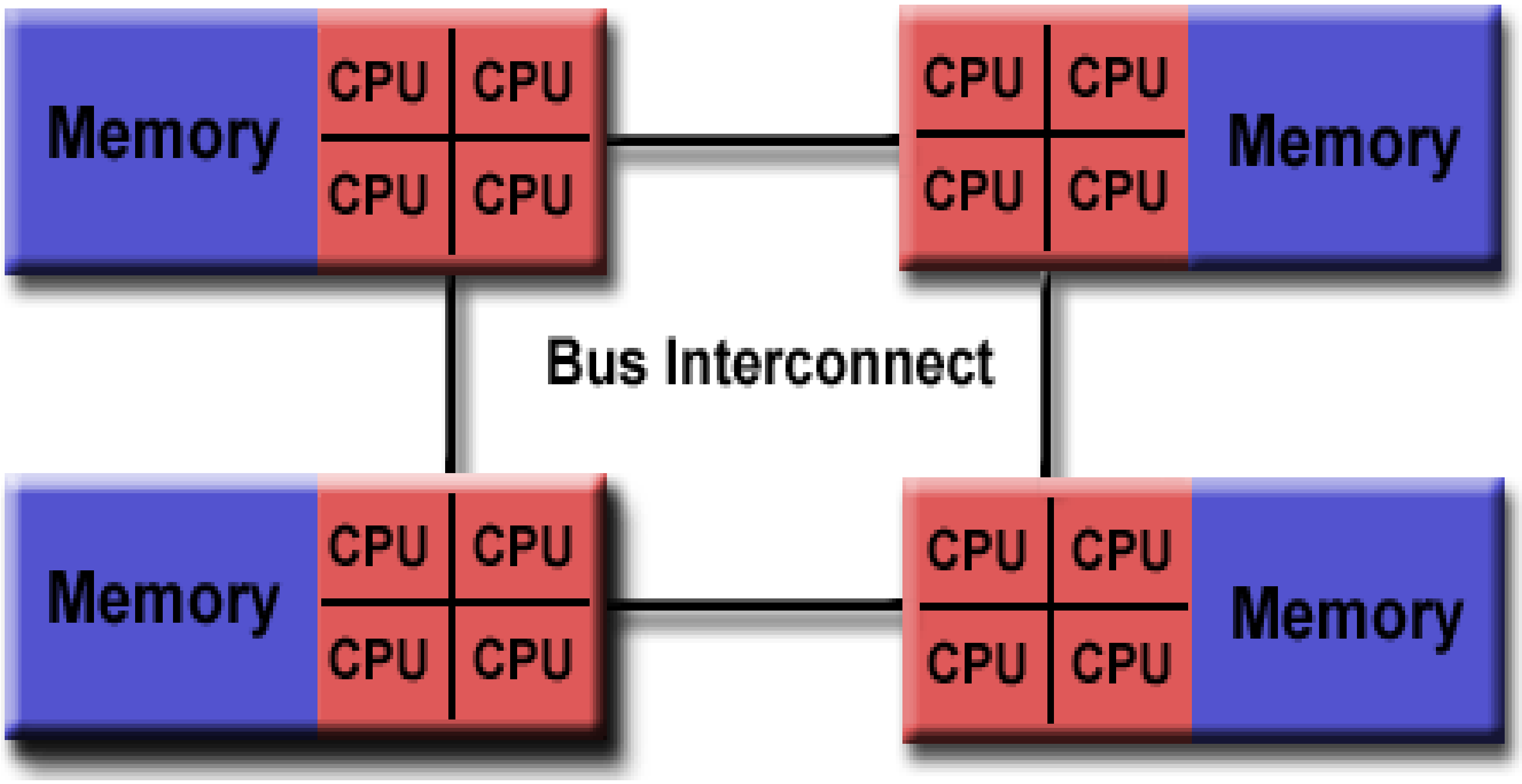
- Standardization: Jointly defined and endorsed by a group of vendors
- Lean and Mean: simple and limited set of directives
- Ease of Use: capability to incrementally parallelize (vs MPI all or nothing approach)
- Portability: C / C++ API for all platforms

OpenMP Shared Memory Model

- OpenMP is designed for multi-processor/core, shared memory machines



OpenMP Shared Memory Model



Thread Based Parallelism

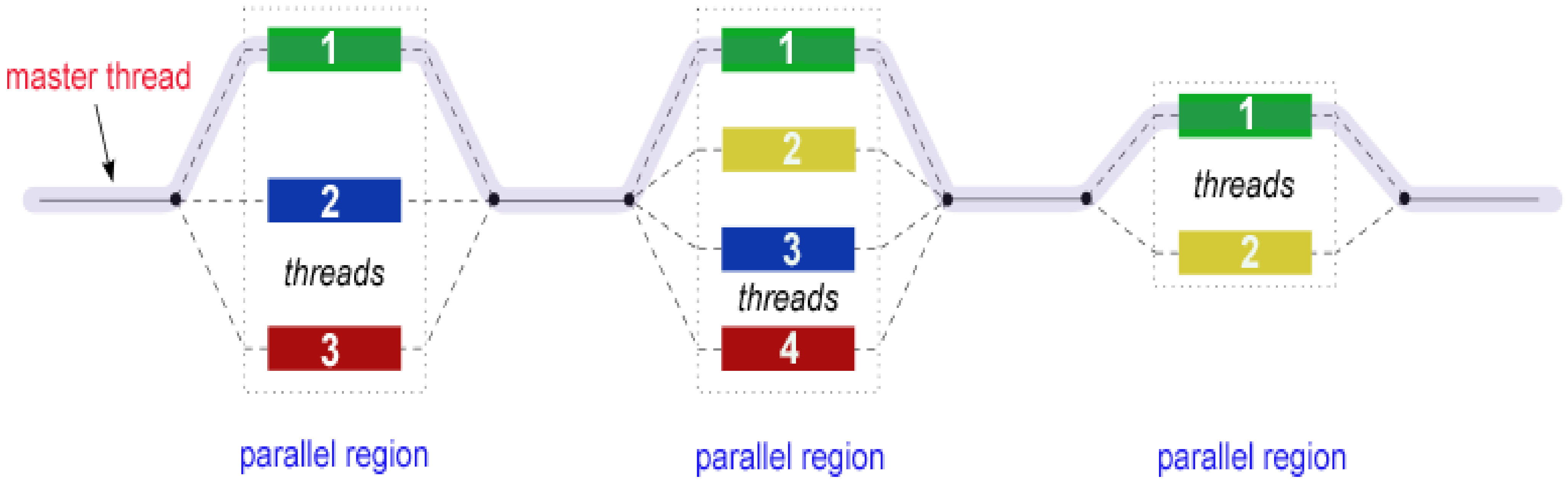
- OpenMP programs accomplish parallelism exclusively through the use of threads
  - A thread of execution is the smallest unit of processing that can be scheduled by an operating system.
  - The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
- Threads exist within the resources of a process.
- Without the process, they cease to exist.
- Typically, the number of threads match the number of machine processors/cores.
- However, the actual use of threads is up to the application

Explicit Parallelism

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization
- Parallelization can be as
  - simple as taking a serial program and inserting compiler directives
  - complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks

Fork – Join Model

- OpenMP uses the fork-join model of parallel execution



Fork – Join Model

- All OpenMP programs begin as a single process: the **master thread**.
- The master thread executes sequentially until the first **parallel region** construct is encountered.
- **Fork:** the master thread then creates a team of parallel ***threads***.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.

Fork – Join Model

- **Join:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary

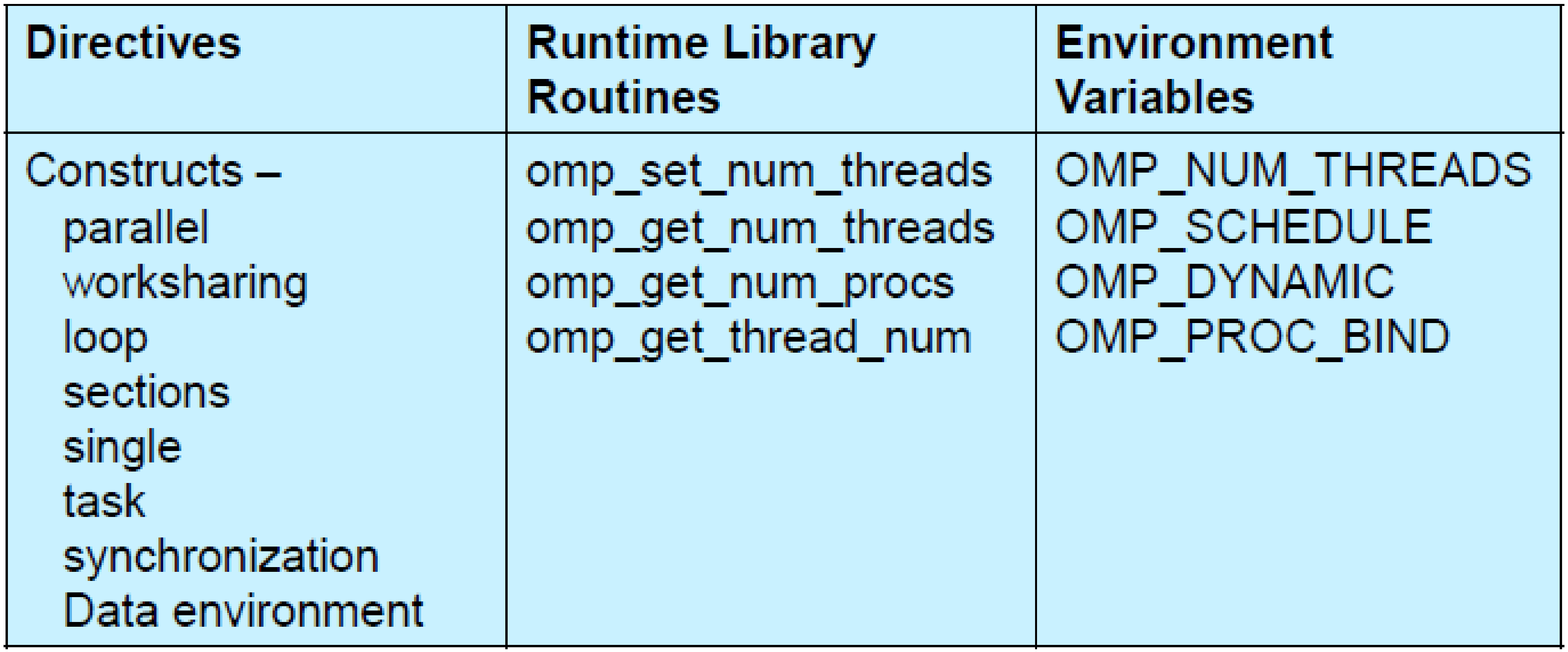
Other Features of OpenMP

- **Compiler Directive Based:** compiler directives which are imbedded in C/C++
- **Nested Parallelism:** placement of parallel regions inside other parallel regions
- **Dynamic Threads:** dynamically alter the number of threads used to execute parallel regions
- **I/O:** specifies nothing about parallel I/O, programmer to ensure that I/O is conducted correctly

Three Components of OpenMP API

- The OpenMP API is comprised of three distinct components:
  - Compiler Directives (44)
  - Runtime Library Routines (35)
  - Environment Variables (13)
- The application developer decides how to employ these components.
- In the simplest case, only a few of them are needed.

Three Components of OpenMP API



Directives	Runtime Library Routines	Environment Variables
Constructs – parallel worksharing loop sections single task synchronization Data environment	omp_set_num_threads omp_get_num_threads omp_get_num_procs omp_get_thread_num	OMP_NUM_THREADS OMP_SCHEDULE OMP_DYNAMIC OMP_PROC_BIND

Compiler Directives

- Compiler directives appear as comments
- OpenMP compiler directives are used for :
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads
- Compiler directives have the following syntax:

Sentinel directive-name [clause, ...]

```
#pragma omp parallel default(shared) private(beta,pi)
```

Run-time Library Routines

- Run-time library routines are used for:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks & nested locks
  - Querying wall clock time and resolution
- For C/C++, all of the run-time library routines are actual subroutines

```
int omp_get_num_threads(void)
```

Environment Variables

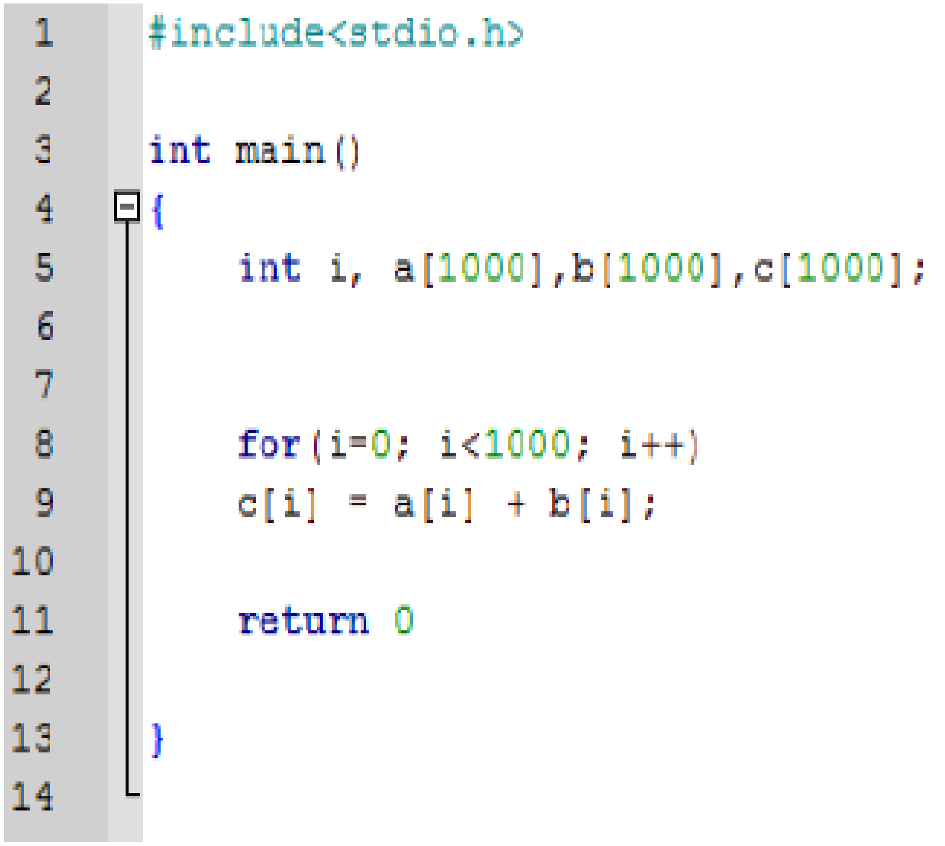
- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
- These environment variables can be used to control:
  - Setting the number of threads
  - Specifying how loop interations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism;
  - Setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size and wait policy

```
export OMP_NUM_THREADS=8
```

General Code Structure

```
#include <omp.h>
main ()
{
  int var1, var2, var3;
  Serial code
      .
      .
  Beginning of parallel region. Fork a team of threads.
  Specify variable scoping
  #pragma omp parallel private(var1, var2) shared(var3)
  {
       Parallel region executed by all threads
       Other OpenMP directives
       Run-time Library calls
       All threads join master thread and disband
   }
   Resume serial code
      .
      .
}
```

Sample C Program



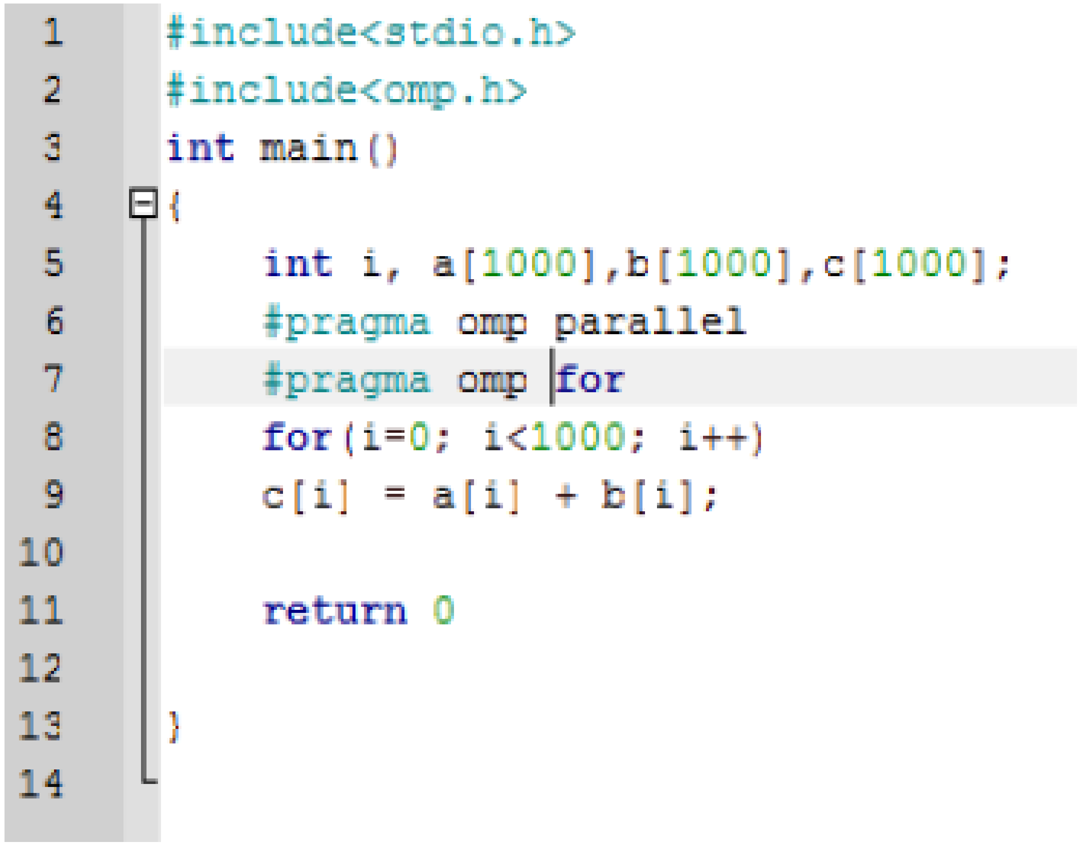
```
#include<stdio.h>

int main()
{
    int i, a[1000],b[1000],c[1000];


    for(i=0; i<1000; i++)
    c[i] = a[i] + b[i];

    return 0

}
```



```
#include<stdio.h>
#include<omp.h>
int main()
{
    int i, a[1000],b[1000],c[1000];
    #pragma omp parallel
    #pragma omp for
    for(i=0; i<1000; i++)
    c[i] = a[i] + b[i];

    return 0

}
```

Compiling

- Include header **<omp.h>**
- All compilers require the use of appropriate compiler flag to "turn on" OpenMP compilations

- **$ gcc –fopenmp filename.c**

OpenMP Directives Format

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

General Rules for Directives

- Case sensitive
- Directives follow conventions of the C/C++ standards for compiler directives
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

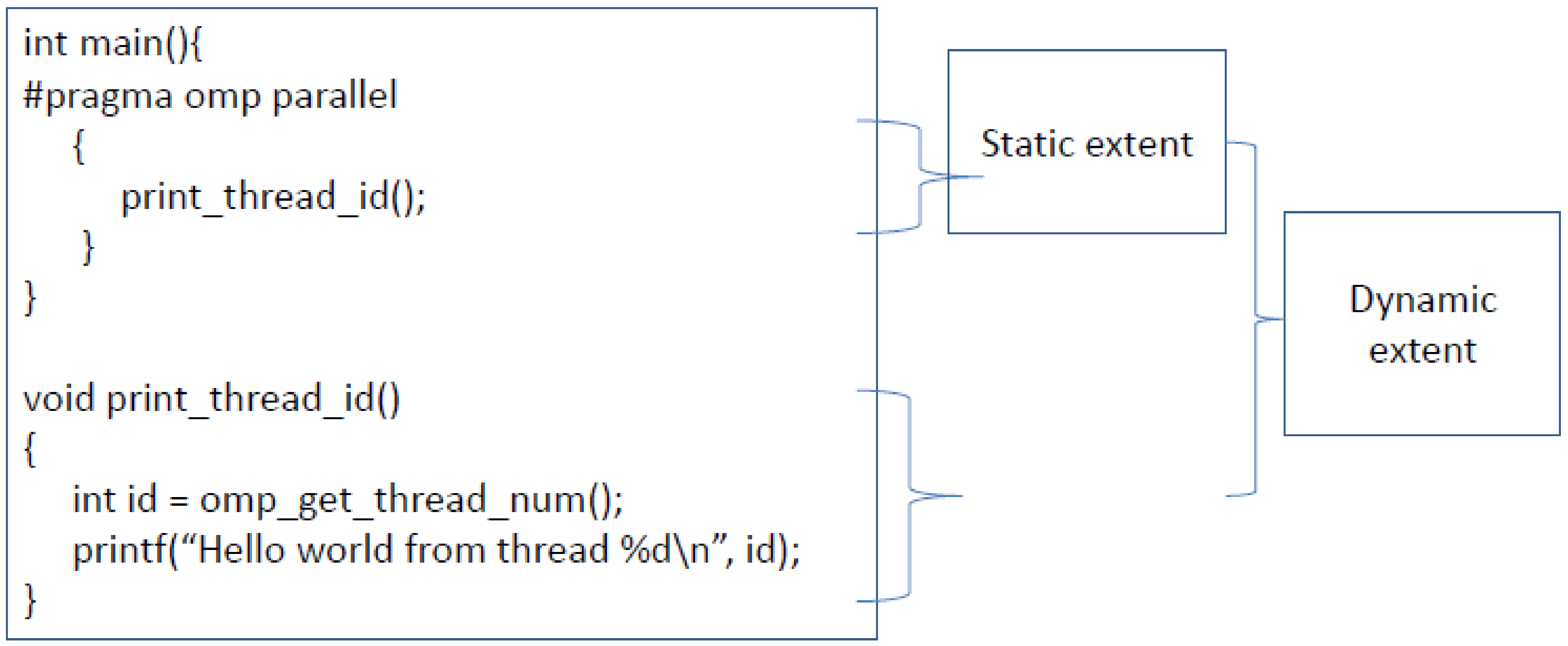
Directive Scoping

- **Static (Lexical) Extent:** The code textually enclosed between the beginning and the end of a structured block following a directive - does not span multiple routines or code files
- **Orphaned Directive:** A directive that appears independently from another enclosing directive is said to be an orphaned directive. It exists outside of another directive's static (lexical) extent.
- **Dynamic Extent:** The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.

Lexical and Dynamic Extents

- Parallel regions enclose an arbitrary block of code, sometimes including calls to another function.
- The lexical or static extent of a parallel region is the block of code to which the parallel directive applies.
- The dynamic extent of a parallel region extends the lexical extent by the code of functions that are called (directly or indirectly) from within the parallel region.
- The dynamic extent is determined only at runtime

Lexical and Dynamic Extents



```
int main(){
#pragma omp parallel
    {
        print_thread_id();
     }
}

void print_thread_id()
{
    int id = omp_get_thread_num();
    printf("Hello world from thread %d\n", id);
}
```

Orphaned Directive

- A directive which is in the dynamic extent of another directive but not in its static extent is said to be orphaned
- Work sharing directives can be orphaned
- This allows a work-sharing construct to occur in a subroutine which can be called both by serial and parallel code, improving modularity

PARALLEL Region Construct

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

PARALLEL Region Construct

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team.
- The master is a member of that team and has thread number 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 region.
- Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

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

Restrictions

- IF clause must evaluate to non-zero value in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.
- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch (goto) into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted
- A program must not depend upon the ordering of the clauses

Parallel Region Example

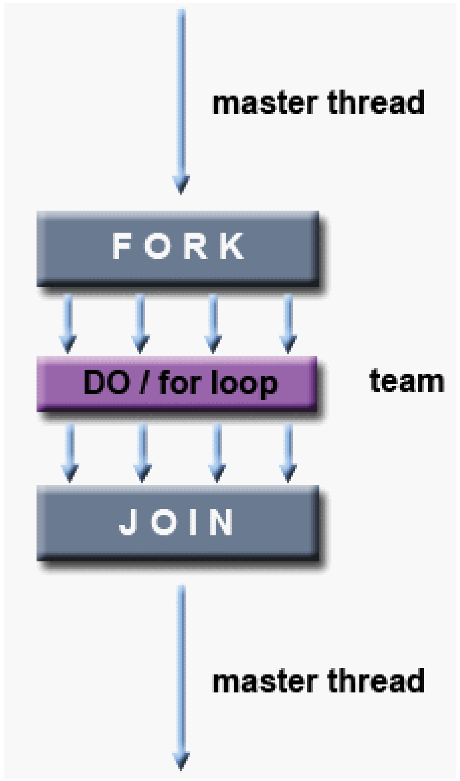
```
#include <omp.h>
main(int argc, char *argv[])
{
    int nthreads, tid;
   /* Fork a team of threads with each thread having a private
                                          tid variable */
    #pragma omp parallel private(tid)
    {
       /* Obtain and print thread id */
         tid = omp_get_thread_num();
         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);
         }
    }  /* All threads join master thread and terminate */
}
```

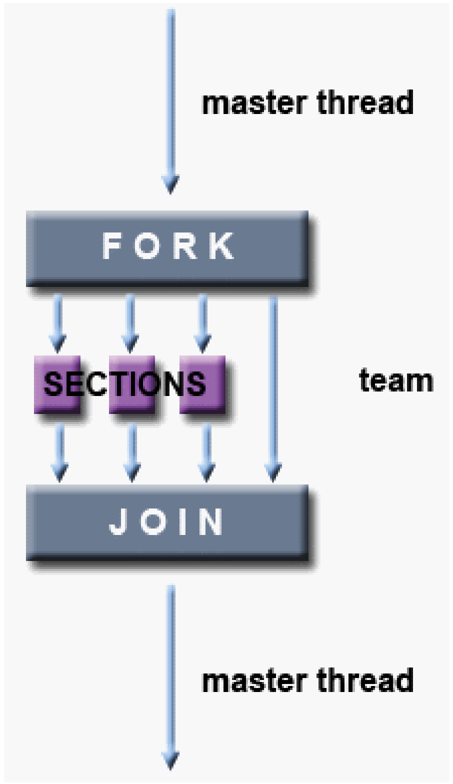
Work-Sharing Constructs

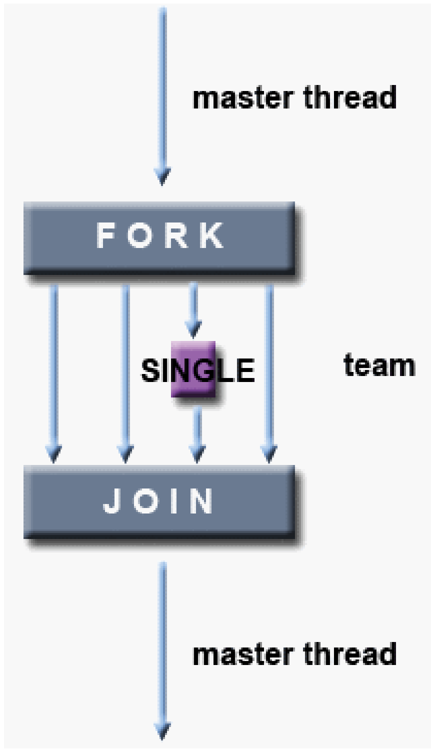
- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

Work-Sharing Constructs

LOOP (for) - shares iterations of a loop across the team. Represents a type of "data parallelism".	SECTION - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".	SINGLE - serializes a section of code







Restrictions

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all
- Successive work-sharing constructs must be encountered in the same order by all members of a team

FOR Directive

- FOR directive specifies that the iterations of the loop immediately following it must be executed in parallel
- This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor

FOR Directive

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

FOR Directive

- SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.
    - STATIC: Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

FOR Directive

- SCHEDULE:
  - DYNAMIC: Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
  - GUIDED: Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.

FOR Directive

- The size of the initial block is proportional to:
  number_of_iterations / number_of_threads

- Subsequent blocks are proportional to
  number_of_iterations_remaining / number_of_threads

- The chunk parameter defines the minimum block size. The default chunk size is 1.
- RUNTIME: The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
- AUTO: The scheduling decision is delegated to the compiler and/or runtime system.

FOR Directive

- **schedule(static [,chunk])**
  - Deal-out blocks of iterations of size “chunk” to each thread.
- **schedule(dynamic[,chunk])**
  - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
- **schedule(guided[,chunk])**
  - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
- **schedule(runtime)**
  - Schedule and chunk size taken from the OMP_ SCHEDULE environment variable

FOR Directive

- **NO WAIT / nowait**: If specified, then 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.

- **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the **schedule** clause.
- The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space

FOR Directive - Restrictions

- It is illegal to branch (goto) out of a loop associated with a **FOR** directive.
- The chunk size must be specified as a loop invariant integer expression, as there is no synchronization during its evaluation by different threads.
- ORDERED, COLLAPSE and SCHEDULE clauses may appear once each.

Sample Program – FOR Directive

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100
main(int argc, char *argv[])
{
   int i, chunk;
   float a[N], b[N], c[N];
   /* Some initializations */
   for (i=0; i < N; i++)
         a[i] = b[i] = i * 1.0;
   chunk = CHUNKSIZE;
   #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 region */
}
```

Sample Program – PARALLEL FOR

```
# include <omp.h>
# define N 1000
# define CHUNKSIZE 100
main(int argc, char *argv[])
{
        int i, chunk;
        float a[N], b[N], c[N];
    /* Some initializations */
        for (i=0; i < N; i++)
            a[i] = b[i] = i * 1.0;
        chunk = CHUNKSIZE;
        #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 region */
}
```

SECTIONS Directive

- The SECTIONS directive is a non-iterative work-sharing construct.
- It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Independent SECTION directives are nested within a SECTIONS directive.
- Each SECTION is executed once by a thread.
- Different sections may be executed by different threads.
- It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.

SECTIONS Directive

```
#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
}
```

SECTIONS Directive - Restrictions

- There is an implied barrier at the end of a SECTIONS directive, unless the NOWAIT / nowait clause is used
- It is illegal to branch (goto) into or out of section blocks.
- SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive (no orphan SECTIONs).

Sample Program … SECTION Directive

```
#include <omp.h>
#define N 1000
main(int argc, char *argv[])
{
    int i;
    float a[N], b[N], c[N], d[N];
    /* Some initializations */
    for (i=0; i < N; i++)
    {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
```

Sample Program … SECTION Directive

```
   #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 region */
}
```

SINGLE Directive

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
- May be useful when dealing with sections of code that are not thread safe (such as I/O)

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

TASK Construct

```
#pragma omp task [clause ...]  newline
                 if (scalar expression)
                 final (scalar expression)
                 untied
                 default (shared | none)
                 mergeable
                 private (list)
                 firstprivate (list)
                 shared (list)
    structured_block
```

Data Scope 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, FOR, and SECTIONS) to control the scoping of enclosed variables.

Data Scope Attribute Clauses

- These constructs provide the ability to control the data environment during execution of parallel constructs.
    - They define how and which data variables in the serial section of the program are transferred to the parallel regions of the program (and back)
    - They define which variables will be visible to all threads in the parallel regions and which variables will be privately allocated to all threads.
- Data Scope Attribute Clauses are effective only within their lexical/static extent.

Data Scope Attribute Clauses

- The PRIVATE clause declares variables in its list to be private to each thread

  private (list)

- The SHARED clause declares variables in its list to be shared among all threads in the team

  shared (list)

- A shared variable exists in only one memory location and all threads can read or write to that address
- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables

Data Scope Attribute Clauses

- The DEFAULT clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region

  default (shared | none)

- The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.

  firstprivate (list)

- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct

Data Scope Attribute Clauses

- The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object

  lastprivate (list)

- The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.

Data Scope Attribute Clauses

- The REDUCTION clause performs a reduction on the variables that appear in its list.
- A private copy for each list variable is created for each thread.
- At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

    reduction (operator: list)

Restrictions on Reduction

- Variables in the list must be named scalar variables.
- They can not be array or structure type variables.
- They must also be declared SHARED in the enclosing context.
- Reduction operations may not be associative for real numbers.
- The REDUCTION clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in statements which have one of following forms:

Sample Program – Reduction

```
#include <omp.h>
main(int argc, char *argv[])
{
     int   i, n, chunk;
     float a[100], b[100], result;
     /* Some initializations */
     n = 100;  chunk = 10;  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);
}
```

Run-Time Library Routines

- OMP_SET_NUM_THREADS: Sets the number of threads that will be used in the next parallel region
- OMP_GET_NUM_THREADS: Returns the number of threads that are currently in the team executing the parallel region from which it is called
- OMP_GET_MAX_THREADS: Returns the maximum value that can be returned by a call to the OMP_GET_NUM_ THREADS function
- OMP_GET_THREAD_NUM: Returns the thread number of the thread, within the team, making this call.
- OMP_GET_THREAD_LIMIT: Returns the maximum number of OpenMP threads available to a program

Run-Time Library Routines

- OMP_GET_NUM_PROCS: Returns the number of processors that are available to the program
- OMP_IN_PARALLEL: Used to determine if the section of code which is executing is parallel or not
- OMP_SET_DYNAMIC: Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions
- OMP_GET_DYNAMIC: Used to determine if dynamic thread adjustment is enabled or not
- OMP_SET_NESTED: Used to enable or disable nested parallelism
- OMP_GET_NESTED: Used to determine if nested parallelism is enabled or not

Run-Time Library Routines Format

```
void omp_set_num_threads
                (int num_threads);
int omp_get_num_threads (void);
int omp_get_max_threads (void);
int omp_get_thread_num (void);
int omp_get_thread_limit (void):
int omp_get_num_procs (void);
int omp_in_parallel(void);
```

OpenMP Library Functions

```
void omp_set_dynamic
            (int dynamic_threads);
int omp_get_dynamic (void);
void omp_set_nested (int nested);
int omp_get_nested (void);
void omp_set_schedule
    (omp_sched_t kind, int modifier)
void omp_get_schedule(omp_sched_t * kind, int *
modifier)
```

Environment Variables

- OpenMP provides environment variables for controlling the execution of parallel code.
- All environment variable names are uppercase.
- The values assigned to them are not case sensitive

- **OMP_SCHEDULE:** Applies only to **for, parallel for** directives which have their schedule clause set to RUNTIME. The value of this variable determines how iterations of the loop are scheduled on processors.

Environment Variables

- **OMP_NUM_THREADS:** Sets the maximum number of threads to use during execution
- **OMP_DYNAMIC:** Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE.
- **OMP_PROC_BIND:** Enables or disables threads binding to processors. Valid values are TRUE or FALSE
- **OMP_NESTED:** Enables or disables nested parallelism. Valid values are TRUE or FALSE.

Synchronization Constructs

- The MASTER directive specifies a region that is to be executed only by the master thread of the team
- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time
- The BARRIER directive synchronizes all threads in the team
- The TASKWAIT construct specifies a wait on the completion of child tasks generated since the beginning of the current task

Synchronization Constructs

- The ATOMIC directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it
- The FLUSH directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point.

Synchronization Constructs

- The ORDERED directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.
  - Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed.
- The THREADPRIVATE directive is used to make global file scope variables, local and persistent to a thread through the execution of multiple parallel regions.

Synchronization Constructs Format

```
#pragma omp barrier

#pragma omp single [clause list]
        structured block
#pragma omp master
#pragma omp critical [(name)]
#pragma omp ordered

#pragma omp atomic
    statement_expression

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