Introduction to OpenMP

Let's read a bit

OpenMP is a library that supports shared memory multiprocessing. The OpenMP programming model is SMP (symmetric multi-processors, or shared-memory processors): that means when programming with OpenMP all threads share memory and data.

Parallel code with OpenMP marks, through a special directive, sections to be executed in parallel. The part of the code that's marked to run in parallel will cause threads to form. The main thread is the master thread. The slave threads all run in parallel and run the same code. Each thread executes the parallelized section of the code independently. When a thread finishes, it joins the master. When all threads finished, the master continues with code following the parallel section.

Each thread has an ID attached to it that can be obtained using a runtime library function (called omp_get_thread_num()). The ID of the master thread is 0.

OpenMP supports C, C++ and Fortran.

The OpenMP functions are included in a header file called

The OpenMP parts in the code are specified using #pragmas

OpenMP has directives that allow the programmer to:

- Specify the parallel region (create threads)
- specify how to parallelize loops
- specify the scope of the variables in the parallel section (private and shared)
- specify if the threads are to be synchronized
- specify how the works is divided between threads (scheduling)

OpenMP hides the low-level details and allows the programmer to describe the parallel code with high-level constructs, which is as simple as it can get.

What is OpenMP?

- An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism Comprises three primary API components
    - Compiler Directives
    - Runtime Library Routines
    - Environment Variables
    - Portable
- The API is specified for C/C++ and Fortran

- Has been implemented for most major platforms including Unix/ Linux platforms and Windows NT
- Standardized
    - Jointly defined and endorsed by a group of major computer hardware and software vendors
    - Expected to become an ANSI standard later???
- What does OpenMP stand for?
    - Short version: Open Multi-Processing
    - Long version: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia

History of openmp

The OpenMP Architecture Review Board (ARB) published its first API specifications, OpenMP for Fortran 1.0, in October 1997. In October the following year they released the C/C++ standard. 2000 saw version 2.0 of the Fortran specifications with version 2.0 of the C/C++ specifications being released in 2002. Version 2.5 is a combined C/C++/Fortran specification that was released in 2005.

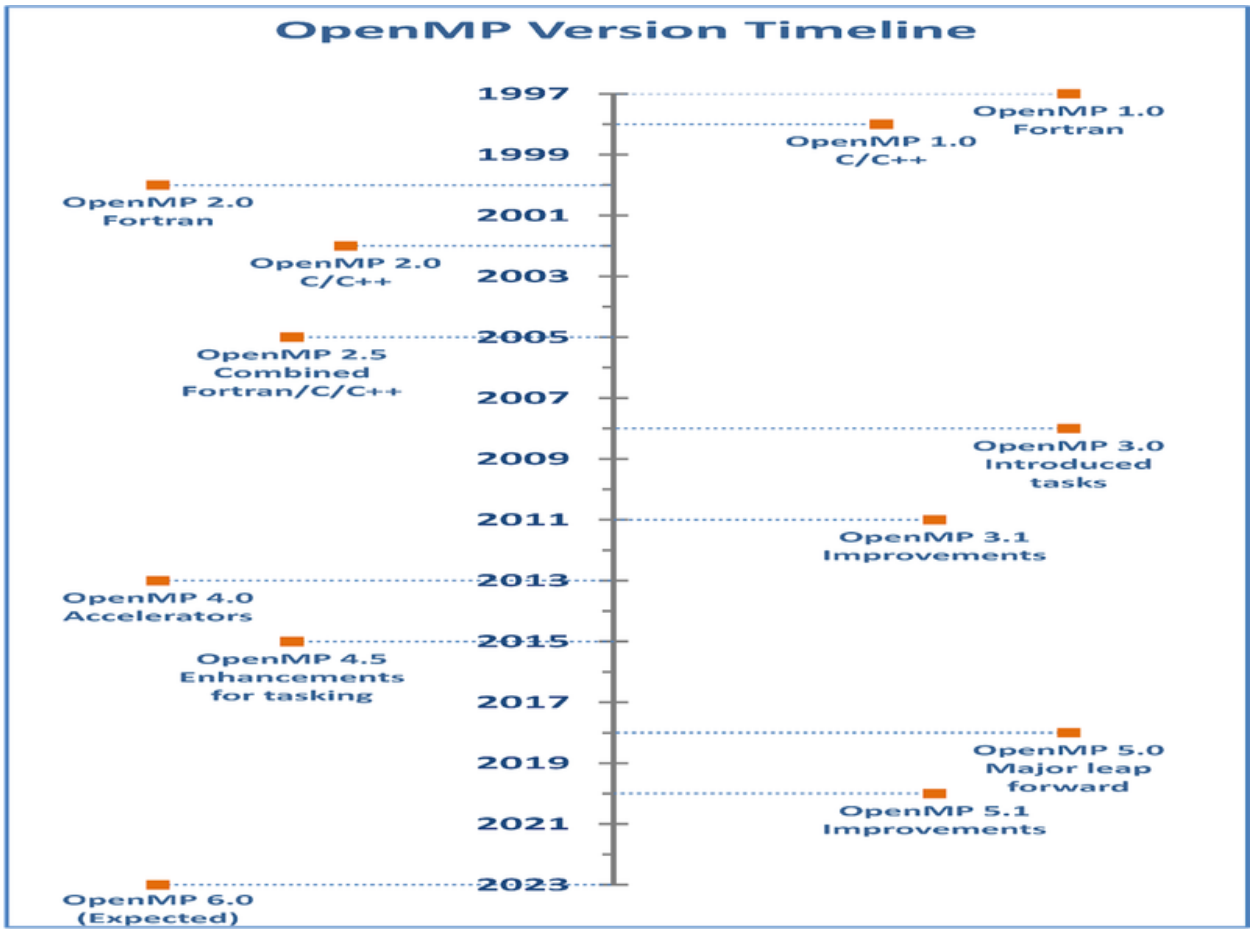
Up to version 2.0, OpenMP primarily specified ways to parallelize highly regular loops, as they occur in matrix-oriented numerical programming, where the number of iterations of the loop is known at entry time. This was recognized as a limitation, and various task parallel extensions were added to implementations. In 2005, an effort to standardize task parallelism was formed, which published a proposal in 2007, taking inspiration from task parallelism features in Cilk, X10 and Chapel.

Version 3.0 was released in May 2008. Included in the new features in 3.0 is the concept of tasks and the task construct, significantly broadening the scope of OpenMP beyond the parallel loop constructs that made up most of OpenMP 2.0.

Version 4.0 of the specification was released in July 2013. It adds or improves the following features: support for accelerators; atomics; error handling; thread affinity; tasking extensions; user defined reduction; SIMD support; Fortran 2003 support.

The current version is 5.2, released in November 2021.

Note that not all compilers (and OS) support the full set of features for the latest version/s



Why Openmp?

- More efficient
- Hides the low-level details.
- OpenMP has directives that allow the programmer to:
    - specify the parallel region
    - specify whether the variables in the parallel section are private or shared
    - specify how/if the threads are synchronized
    - specify how to parallelize loops
    - specify how the works is divided between threads (scheduling)
- Shared memory, thread-based parallelism
    - OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm.
    - A shared memory process consists of multiple threads.
- Explicit Parallelism
    - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

- OpenMP uses the fork-join model of parallel execution.
- Compiler directive based
    - -Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

OpenMP Programming Model

Fork–join model:

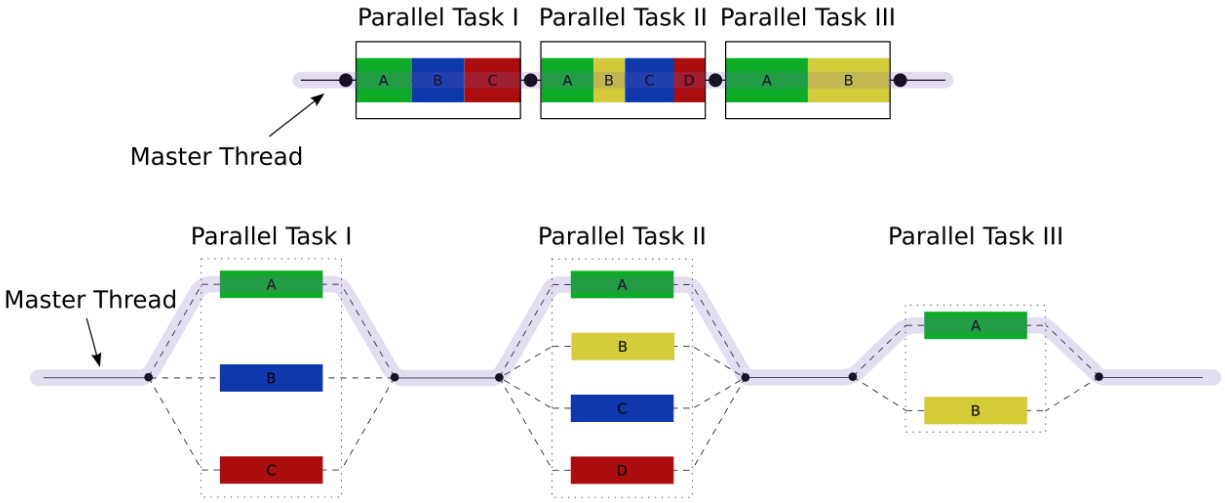
The fork-join model is a parallel programming model that is used to speed up the execution of a program by breaking it down into smaller pieces that can be executed in parallel. The model is based on the idea of dividing a task into smaller sub-tasks, which can be executed concurrently, and then joining the results of those sub-tasks to produce the final result.

The fork-join model consists of three main components: the fork operation, the join operation, and the task.

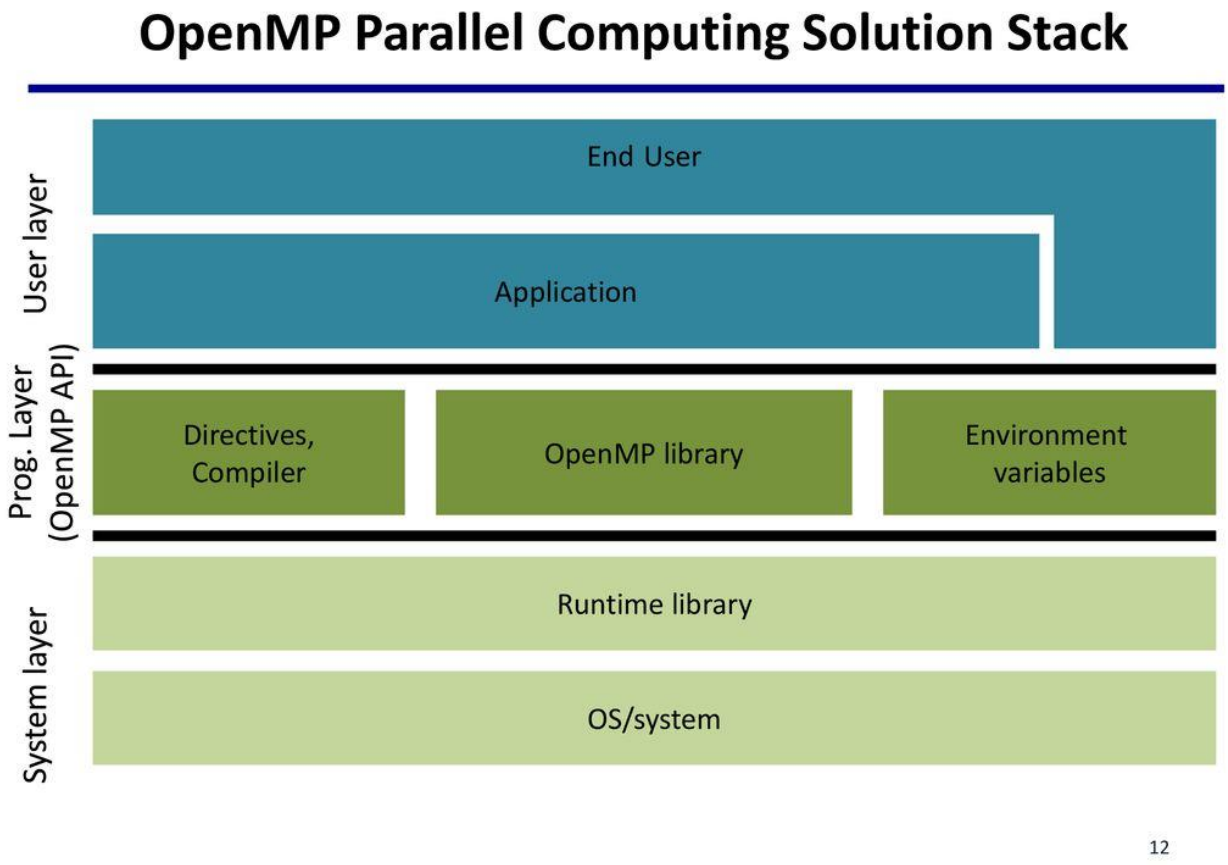
1. The fork operation splits a task into smaller sub-tasks and assigns each sub-task to a separate thread or process for execution.
2. The join operation waits for all the sub-tasks to complete and then combines their results to produce the final result.
3. The task is the unit of work that is being executed and can be further subdivided into smaller sub-tasks as needed.

The fork-join model is particularly well-suited for parallelizing recursive algorithms, such as the quicksort algorithm. In these algorithms, the task is split into smaller sub-tasks, which are then sorted independently and combined to produce the final sorted result.

Java's Fork/Join framework is a popular implementation of the fork-join model, which provides a simple way to implement parallel algorithms using the fork-join model.



Openmp Stack



- System layer:
  - OS
  - Runtime library
- Programming layer:

  Openmp API

  - Compiler Directive
  - Openmp library
  - Environment variables
- User layer:
  - Application
  - End user

Hello World In OpenMP

we can see a c program which prints hello world on console

create a file:

vi file_name.c

Wrilte the below program in the file.

```
#include <stdio.h>
int main()
{
  // printf() displays the string inside quotation
  printf("Hello, World!");
  return 0;
}
```

Compilation and Execution

gcc file_name.c -o file_name

./file_name

Note: file_name can be any name that you want

Output:

Hello, World!

This how we write compile and execute normal c programs.

let`s make the above program for openmp by following basic syntax.

Basic Syntax

- **Function prototypes and types in the file:**

```
#include <omp.h>
```

- **Most of the constructs in OpenMP are compiler directives.**

```
#pragma omp construct [clause [clause]…]
        {
                //..Do some work here
```

```
}
 //end of parallel region/block
```

- **Example:**

```
#pragma omp parallel num_threads(4)
```

OpenMP Hello World program using C

This how a hello world program in openmp look likes

```
#include<stdio.h>
#include<omp.h>
int main()
{
    #pragma omp parallel
    {
    int ID = omp_get_thread_num();
    printf("Hello, world(%d)\n",ID);
    }
  return 0;
}
```

- The Expected output of the given program is it will print the hello world with there thread id and the number of prints of hello world is equal to the number of thread which are set before the execution of program.
- The 1st line of program includes the header file in c which has necessary information to include the input/output related functions in our program.

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

- The 2nd line of program includes the header file in c for openmp API

  ```
  #include<omp.h>
  ```

- main method contain the all part of program that will be executed how the instructions written in it

  ```
  int main()
  {
  ```

```
        //Block of code
}
```

- In main method the given line is compiler directive which Specify the parallel region we need to mention the region which we are going to make it as parallel using the keyword pragma omp parallel. The pragma omp parallel is used to fork additional threads to carry out the work enclosed in the parallel. The original thread will be denoted as the master thread with thread ID 0.

```
#pragma omp parallel
{
        //code block
}
```

- The omp_get_thread_num() is the openmp library routine which returns the thread number, within the current team.

```
int ID = omp_get_thread_num();
```

Compile and execution using GCC compiler:

Compile : icc -qopenmp file_name.c -o file_name

Set the number of Threads: export OMP_NUM_THREADS=4

Execution: ./file_name

Compile and execution using Intel compiler:

Compile : icc -qopenmp file_name.c -o file_name

Set the number of Threads: export OMP_NUM_THREADS=4

Execution: ./file_name

Output:

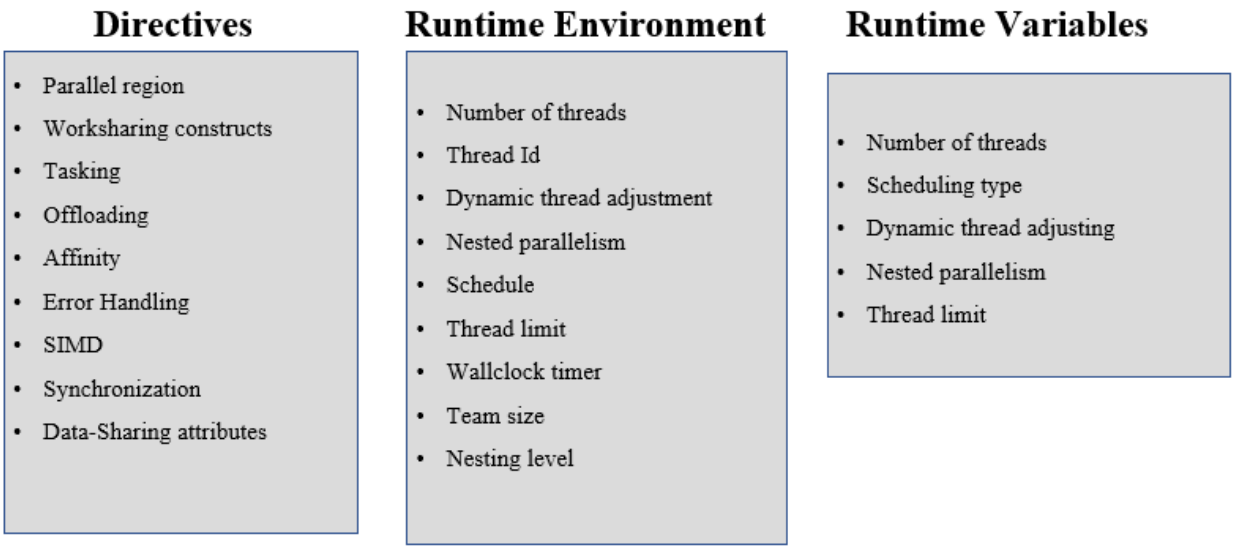
Hello, world(0)

Hello, world(2)

Hello, world(1)

Hello, world(3)

OpenMP Components



Directives

Directives in OpenMP are special instructions that are used to indicate to the compiler how to parallelize certain parts of a program. These directives are inserted into the source code of a program as special comments or #pragma statements. When the program is compiled, the OpenMP compiler recognizes these directives and generates code that can be executed in parallel on a shared memory system.

Directives in OpenMP provide a high-level way to express parallelism in a program. They can be used to specify which parts of a program should be executed in parallel, how many threads should be used to execute the parallel region, and how the data should be shared or synchronized between threads.

OpenMP provides a wide range of directives that can be used to express parallelism in various parts of a program. For example, the #pragma omp parallel directive is used to specify a region of code that should be executed in parallel by multiple threads. The #pragma omp for directive is used to parallelize loops by distributing the loop iterations across the threads in a team.

Other OpenMP directives are used to control the synchronization and communication between threads. For example, the #pragma omp barrier directive is used to ensure that all threads in a parallel region have completed their work before continuing. The #pragma omp critical directive is used to specify a section of code that must be executed by only one thread at a time to avoid race conditions.

Parallel Region Construct/Directives

- Block of code that will be executed by multiple threads
- Fundamental OpenMP parallel 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 section 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.
- The number of threads in a parallel region is determined by the following factors, in order of precedence:

1. Evaluation of the IF clause
2. Setting of the NUM_THREADS clause
3. Use of the omp_set_num_threads() library function
4. Setting of the OMP_NUM_THREADS environment variable
5. Implementation default - usually the number of cores on a node.

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

#pragma omp parallel [clause[[,]clause] ...] new-line

clause:

if(scalar-expression)

num_threads(integer-expression)

default(shared | none)

private(list)

firstprivate(list)

shared(list)

reduction(operator: list)

If clause

In OpenMP, the "if" clause can be used to conditionally control the execution of a parallel region. The syntax for the "if" clause is as follows:

```
#pragma omp parallel if(condition)
{
```

```
    // code to be executed in parallel
}
```

Here, the "condition" is an expression that must evaluate to either true or false. If the condition is true, the parallel region will be executed in parallel by the threads. If the condition is false, the region will not be executed in parallel, and instead, only the master thread will execute the code within the parallel region.

For example, consider the following code snippet:

```
#include <omp.h>
#include <stdio.h>
int main ()
{
   #pragma omp parallel if(omp_in_parallel)
   {
      printf("The threads is %d\n",omp_get_thread_num());
   }
   printf("The threads is %d\n",omp_get_thread_num());
   return 0;
}
```

In this code, we have a parallel region with an "if" clause that checks if the the parallel region is set or not, the parallel region will not be executed in parallel, and instead, only the master thread will execute the code within the parallel region. Therefore, the output of this code will be:

The threads is 0

If the condation is true then, the parallel region will be executed in parallel by all threads, and the output will be if we are using 4 threads:

```
The threads is 0
The threads is 2
The threads is 1
The threads is 3
```

Nested Parallel Regions

In OpenMP, a parallel nested region is a construct in which a parallel section of code is divided into smaller parallel sections, each of which can be further divided into even smaller parallel sections.

Here is an example of a parallel nested region in OpenMP:

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
   #pragma omp single
   {
      printf("Level %d: number of threads in the team - %d\n",
            level, omp_get_num_threads());
   }
 }
int main()
{
   omp_set_dynamic(0);
   #pragma omp parallel num_threads(2)
   {
      report_num_threads(1);
      #pragma omp parallel num_threads(2)
      {
         report_num_threads(2);
         #pragma omp parallel num_threads(2)
         {
            report_num_threads(3);
         }
      }
   }
   return(0);
```

}

- **Use the omp_get_nested() library function to determine**

  if nested parallel regions are enabled.

- **The two methods available for enabling nested parallel regions (if supported) are:**
  - **The omp_set_nested() library routine**
  - **Setting of the OMP_NESTED environment variable to TRUE**
  - **If not supported, a parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.**
- **Dyanmic Mode**
  - **The number of threads can differ between parallel region of the same program**
  - **The specified number of the threds is an upper bound - the actual number of threds can be smaller to optimize system resources.**
- **Static Mode**
  - **The number of threds for the parallel region is fixed and exactly equal to what the programmer specifies.**

output:

```
(base) [cdacapp@login04 nested]$ export OMP_NESTED=True
(base) [cdacapp@login04 nested]$ ./nested
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
(base) [cdacapp@login04 nested]$
```

Work-sharing Constructs

Work-sharing constructs are used in parallel programming to distribute work among multiple threads in a shared-memory system. These constructs allow multiple threads to work on different parts of a shared memory region simultaneously, improving performance and reducing execution time.

- 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.
- 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.
- DO / for
    - Shares iterations of a loop across the team
    - Represents a type of "data parallelism”
- SECTIONS
    - 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

For Directive/Construct

The for construct tells OpenMP that the iteration set of the for loop that follows is to be distributed across the threads present in the team. Without the for construct, the entire iteration set of the for loop concerned will be executed by each thread in the team.

#pragma omp for [clause[[,] clause] ...] new-line

for-loops

clause:

private(list)

firstprivate(list)

lastprivate(list)

reduction(operator: list)

schedule(kind[, chunk_size])

ordered

nowait

For loop - PRIVATE

In OpenMP, the private clause is used to declare variables that should have private copies for each thread in a team. This is often used in loops to avoid race conditions and ensure thread safety.

The private clause can be used with the #pragma omp for directive to declare variables as private to each thread that executes a subset of the loop iterations. For example:

```
#include<stdio.h>
#include<omp.h>
#define N  5
int main()
{
   int a = 10; //shared
   int b = 20; //shared
   int c = 20;
   omp_set_num_threads(N);
   #pragma omp parallel for private(c)
   for(int i = 0; i < N;i++){
      printf("The thred %d value is %d\n",omp_get_thread_num(),c);
      c = a + b;    //private
      printf("The thred %d value is %d\n",omp_get_thread_num(),c);
   }
   return 0;
}
```

In this example, the private clause declares the variable c as private to each thread that executes a subset of the loop iterations. Each thread will have its own private copy of the variable, avoiding race conditions and ensuring thread safety.

For loop - SCHEDULE

The schedule clause tells OpenMP how to distribute the loop iterations to the threads.

The OpenMP scheduling kind to use.

Possible values:

- auto: the auto scheduling kind will apply.

- dynamic: the dynamic scheduling kind will apply.
- guided: the guided scheduling kind will apply.
- runtime: the runtime scheduling kind will apply.
- static: the static scheduling kind will apply.

schedule(static, [n])

- Each thread is assigned chunks in “round robin” fashion, known as block cyclic scheduling.
- divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread
- If n has not been specified, it will contaN

CEILING(number_of_iterations / number_of_threads) iterations

Example: loop of length 16, 3 threads, chunk size 2 :

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char* argv[])
{
    omp_set_num_threads(3);
    printf("With no chunksize passed:\n");
    #pragma omp parallel for schedule(static)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }
    printf("With a chunksize of 2:\n");
    #pragma omp parallel for schedule(static, 2)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }
    return 0;
```

```
}
```

For loop - SCHEDULE(dynamic)

schedule(dynamic, [n])

Iteration of loop are divided into chunks containing n iterations each.

Default chunk size is 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char* argv[])
{
    omp_set_num_threads(2);
    printf("With no chunksize passed:\n");
    #pragma omp parallel for schedule(dynamic)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }
    printf("With a chunksize of 2:\n");
    #pragma omp parallel for schedule(dynamic, 2)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

For loop - SCHEDULE(guided, [n])

- If you specify n, that is the minimum chunk size that each thread should posses..

- Size of each successive chunk is exponentially decreasing.
- Initial chunk size

  max(number_of_iterations / number_of_threads,n)

  Subsequent chunk consist of

  max(remaining_iterations/number_of_threds,n) iterations

  Schedule(runtime):export OMP_SCHEDULE “STATIC,4”

- Determine the scheduling type at run time by the OMP_SCHEDULE environment variable.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    #pragma omp parallel for schedule(guided)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

For loop - SCHEDULE(runtime)

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char* argv[])
```

```
{
    omp_set_num_threads(4);
    #pragma omp parallel for schedule(runtime)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

SECTIONS Construct

- The sections construct is a non-iterative worksharing construct that contains a set of structured blocks.
- The block are distributed among and executed by the threads in a team.
- Each structured block is executed once by one of the threads in the team in the context of its implicit task.

The parallel directive supports the following clauses:

- **private(list)**
- **firstprivate(list)**
- **lastprivate([modifier :] list)**
- **reduction([modifier,] identifier : list)**
- **nowait**

SECTIONS - PRIVATE

```
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i;
```

```
float a[N], b[N], c[N], d[N],e[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
/* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,chuk */
#pragma omp parallel shared(a,b,c,d,e) private(i)
  {
  #pragma omp sections
    {
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("section 1 # Working thread : %d | %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
    #pragma omp section
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("section 2 # Working thread : %d | %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }  /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

SECTIONS - FIRSTPRIVATE

```
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i;
float a[N], b[N], c[N], d[N],e[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
/* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,chuk */
#pragma omp parallel shared(a,b,c,d,e) firstprivate(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("section 1 # Working thread : %d | %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
    #pragma omp section
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("section 2 # Working thread : %d | %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
```

```
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] - b[i];
      printf("section 3 # Working thread : %d | %f - %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] / b[i];
      printf("section 4 # Working thread : %d | %f / %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }

    }  /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

SECTIONS – LASTPRIVATE

```
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i;
float a[N], b[N], c[N], d[N],e[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
```

```
  }
/* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,chuk */
#pragma omp parallel shared(a,b,c,d,e)
  {
  #pragma omp sections nowait lastprivate(i)
    {
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("section 1 # Working thread : %d | %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
    #pragma omp section
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("section 2 # Working thread : %d | %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }  /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

SECTIONS - NOWAIT

```
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
```

```
int i;
float a[N], b[N], c[N], d[N],e[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
/* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,chuk */
#pragma omp parallel shared(a,b,c,d,e) private(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("section 1 # Working thread : %d | %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
    #pragma omp section
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("section 2 # Working thread : %d | %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }  /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

SINGLE Construct

single is a clause that must be used in a parallel region; it tells OpenMP that the associated block must be executed by one thread only, .

The clause to appear on the single construct, which is one of the following:

- private(list)
- firstprivate(list)
- copyprivate(list)
- nowait

master will be executed by the master only while single can be executed by whichever thread reaching the region first.

SINGLE-PRIVATE

```
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
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;
  }
  /* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,d */
#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp single

    {
```

```
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("1st loop # Working thread : %d # %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
     }
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("2nd loop # Working thread : %d # %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }
    /* end of sections */
  }  /* end of parallel section */
          return 0;
}
```

SINGLE-PRIVATE

```
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i = 1;
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;
  }
```

```
  /* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,d */
#pragma omp parallel shared(a,b,c,d) firstprivate(i)
  {
  #pragma omp single


    {
   for (i=0; i < N; i++){
     c[i] = a[i] + b[i];
     printf("1st loop # Working thread : %d # %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
   for (i=0; i < N; i++){
     d[i] = a[i] * b[i];
     printf("2nd loop # Working thread : %d # %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
   }
   }
   /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

SINGLE-FIRSTPRIVATE

```
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i = 1;
```

```
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;
  }
  /* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,d */
#pragma omp parallel shared(a,b,c,d) firstprivate(i)
  {
  #pragma omp single


    {
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
     printf("1st loop # Working thread : %d # %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
     }
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
     printf("2nd loop # Working thread : %d # %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }
   /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

SINGLE-COPYPRIVATE

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char* argv[])
{
	int a = 123;

	#pragma omp parallel default(none) firstprivate(a)
	{
		printf("Thread %d: a = %d.\n", omp_get_thread_num(), a);

		#pragma omp barrier

		#pragma omp single copyprivate(a)
		{
			a = 456;
			printf("Thread %d executes the single construct and changes a to %d.\n", omp_get_thread_num(), a);
		}

		printf("Thread %d: a = %d.\n", omp_get_thread_num(), a);
	}

	return 0;
}
```

SINGLE-NOWAIT

```
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
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;
  }
  /* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,d */
#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp single nowait


    {
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("1st loop # Working thread : %d # %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
     }
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("2nd loop # Working thread : %d # %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
```

```
    }
    /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

Synchronization Constructs

Synchronization constructs are mechanisms used in computer programming to manage the access to shared resources in a multi-threaded or multi-process environment. These constructs help prevent race conditions, deadlocks, and other synchronization issues that can arise when multiple threads or processes access shared resources concurrently.

Here are some common synchronization constructs used in computer programming:

1. Locks: A lock is a mechanism that allows only one thread or process to access a shared resource at a time. When a thread or process acquires a lock, it gains exclusive access to the resource until it releases the lock.
2. Semaphores: A semaphore is a synchronization construct that maintains a count of the number of available resources. Threads or processes can acquire or release resources by decrementing or incrementing the semaphore count, respectively. If the count is zero, the semaphore will block until a resource becomes available.
3. Monitors: A monitor is a synchronization construct that provides a high-level abstraction for managing shared resources. Monitors allow threads to wait for a resource to become available and signal other threads when a resource is released. Monitors also ensure that only one thread can access the resource at a time.
4. Mutexes: A mutex is a synchronization construct that provides mutual exclusion between threads or processes. A mutex ensures that only one thread or process can execute a critical section of code at a time.
5. Barriers: A barrier is a synchronization construct that forces threads or processes to wait until all participating threads or processes have reached a certain point in their execution. Once all threads or processes have reached the barrier, they can continue executing.
6. Read-Write Locks: A read-write lock is a synchronization construct that allows multiple threads to read a shared resource simultaneously but only one thread to write to the resource at a time. This can improve performance in situations where there are many more reads than writes.

Overall, synchronization constructs are essential tools for managing shared resources in multi-threaded or multi-process environments. By using these constructs, programmers can ensure that their code is safe, reliable, and performs well in a concurrent environment.

Critical Construct

A critical construct is a synchronization mechanism used in computer programming to ensure that a block of code is executed atomically, that is, without interruption by other threads or processes. A critical construct is also known as a critical section, mutual exclusion, or mutex.

In a multi-threaded environment, it is possible for two or more threads to access a shared resource simultaneously, leading to race conditions, deadlocks, or other synchronization issues. To prevent these problems, a critical construct is used to define a section of code that can only be executed by one thread or process at a time.

Here is an example of a critical construct in C:

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

int main(int argc, char* argv[])
{
    // Use 4 threads when creating OpenMP parallel regions
    omp_set_num_threads(4);

    int total = 0;

    // Create the parallel region
    #pragma omp parallel default(none) shared(total)
    {
        // Calculate my factorial
        int my_value = 1;
        for(int i = 2; i <= omp_get_thread_num(); i++)
        {
            my_value *= i;
        }
```

```
            // Add my value to the total
            #pragma omp critical
            {
                printf("[Thread %d] Total before I add my value (%d): %d.\n", omp_get_thread_num(), my_value, total);
                total += my_value;
                printf("[Thread %d] Total after I added my value: %d.\n", omp_get_thread_num(), total);
            }
        }

        return 0;
}
```

Master Construct

In OpenMP, the master construct is a synchronization mechanism that is used to specify a block of code that should be executed only by the master thread in a parallel region. The master thread is typically the thread with thread ID 0, although this can be specified using the omp_set_num_threads() function.

Here's an example of how the master construct works in OpenMP:

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

int main(int argc, char* argv[])
{
    // Use 4 threads when creating OpenMP parallel regions
    omp_set_num_threads(4);

    // Create the parallel region
```

```
    #pragma omp parallel
    {
        printf("[Thread %d] Every thread executes this printf.\n", omp_get_thread_num());

        #pragma omp barrier

        #pragma omp master
        {
            printf("[Thread %d] Only the master thread executes this printf, which is me.\n", omp_get_thread_num());
        }
    }

    return 0;
}
```

BARRIER Construct

The BARRIER construct is a synchronization mechanism used in parallel programming to ensure that all threads in a parallel region have completed their current tasks before proceeding to the next set of tasks. The BARRIER construct is often used to coordinate the execution of multiple threads to avoid race conditions and ensure correct program behavior.

Here's an example of how the BARRIER construct works in OpenMP:

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

int main(int argc, char* argv[])
{
    // Use 4 threads when we create a parallel region
    omp_set_num_threads(4);
```

```
    // Create the parallel region
    #pragma omp parallel
    {
        // Threads print their first message
        printf("[Thread %d] I print my first message.\n", omp_get_thread_num());

        // Make sure all threads have printed their first message before moving on.
        #pragma omp barrier

        // One thread indicates that the barrier is complete.
        #pragma omp single
        {
            printf("The barrier is complete, which means all threads have printed their first message.\n");
        }

        // Threads print their second message
        printf("[Thread %d] I print my second message.\n", omp_get_thread_num());
    }

    return 0;
}
```

ATOMIC Construct

The atomic construct is a synchronization mechanism used in parallel programming to ensure that a particular memory location is updated atomically, that is, without interference from other threads that may also be updating the same memory location concurrently. In other words, the atomic construct ensures that the memory operation is executed as a single indivisible unit, preventing race conditions and ensuring correct program behavior.

Here's an example of how the atomic construct works in OpenMP:

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

int main(int argc, char* argv[])
{
    // Use 4 threads when creating OpenMP parallel regions
    omp_set_num_threads(4);
    int total = 0;
    // Create the parallel region
    #pragma omp parallel default(none) shared(total)
    {
        for(int i = 0; i < 10; i++)
        {
            // Atomically add one to the total
            #pragma omp atomic
            total++;
        }
    }
    printf("Total = %d.\n", total);
    return 0;
}
```

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.

- Necessary to instruct the compiler that a variable must be written to/read from the memory system, i.e. that a variable cannot be kept in a local CPU register
    - Keeping a variable in a register in a loop is very common when producing efficient machine language code for a loop

C/C++: #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. For pointers in the list, the pointer itself is flushed, not the object to which it points.
- Implementations must ensure any prior modifications to thread-visible variables are visible to all threads after this point; i.e., compilers must restore values from registers to memory, hardware might need to flush write buffers, etc.
- The FLUSH directive is implied for the directives shown in the table below. The directive is not implied if a NOWAIT clause is present

C/C++

barrier

parallel – upon entry and exit

critical – upon entry and exit

ordered – upon entry and exit

for – upon exit

sections – upon exit

single – upon exit

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 yet.
- Used within a DO / for loop with an ORDERED clause
- The ORDERED directive provides a way to "fine tune" where ordering is to be applied within a loop. Otherwise, it is not required.
- An ORDERED directive can only appear in the dynamic extent of the following directives:
- DO or PARALLEL DO (Fortran)

- for or parallel for (C/C++)
- Only one thread is allowed in an ordered section at any time
- It is illegal to branch into or out of an ORDERED block.
- An iteration of a loop must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.
- A loop that contains an ORDERED directive must be a loop with an ORDERED clause.

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

int main(void)
{
#pragma omp parallel for ordered
for (int i = 1; i <= 8; i++)
{
   #pragma omp ordered
   printf("Thread %d is executing iteration %d \n", omp_get_thread_num(), i);
}
   return 0;
}
```