Course name: High Performance Computing

Course code: DSE 3122

No of contact hours/week: 3

Credit: 3

By

Dr. Sandhya Parasnath Dubey and Dr. Manisha

Assistant Professor

Department of Data Science and Computer Applications - MIT,

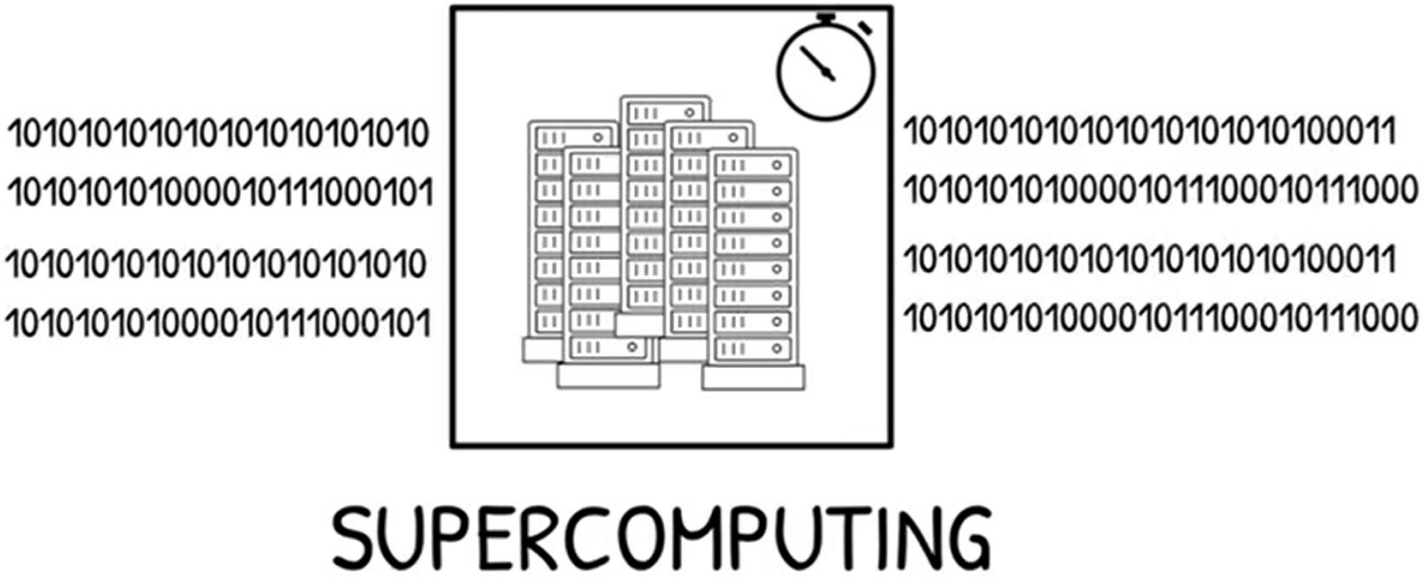
Manipal Academy of Higher Education, Manipal-576104, Karnataka, India.

Email: sandhya.dubey@manipal.edu; manisha.mit@manipal.edu

Objectives

- To introduce you to the basic concepts and ideas in parallel computing.
- To familiarize you with the major programming models in parallel computing.
- To provide you with guidance for designing efficient parallel programs.

What is High Performance Computing (HPC)?



- Use of supercomputers and parallel processing techniques to solve complex computational problems.

Why HPC is needed?

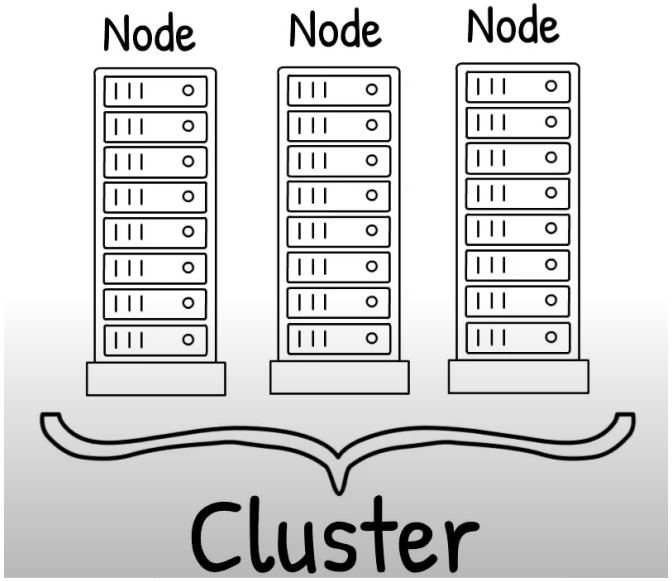
- If you are processing more data than what your resources can handle and you have to wait for weeks or months to see the results.

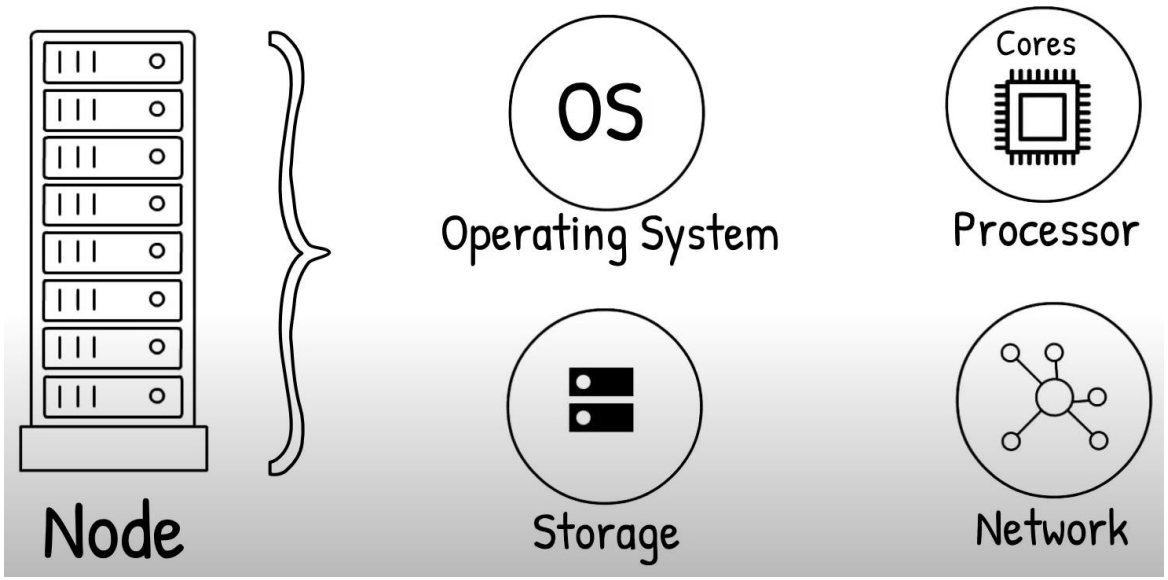
- In data science and engineering, HPC is essential for processing large datasets, running complex simulations, and analyzing results quickly and efficiently.

Key components of HPC systems

- **Powerful Processors**: Multiple CPUs or GPUs working together to perform many calculations at a time.
- **Large Memory**: To handle extensive data sets.
- **Fast Storage**: To quickly read and write data.
- **High-speed Networking**: To connect different parts of the system quickly and efficiently.

Architecture of HPC systems





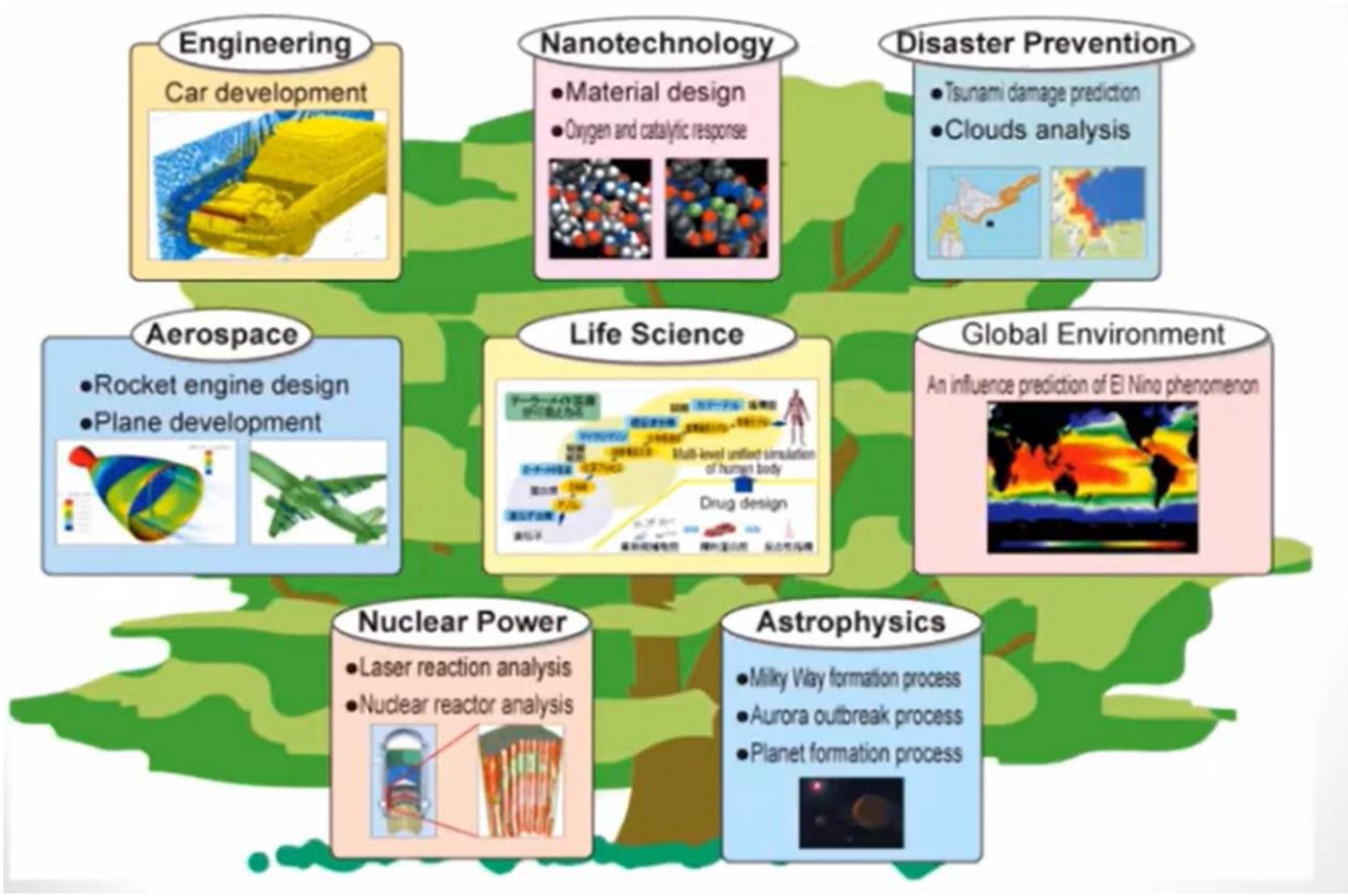
Basic Concepts in HPC?

- **Parallel Computing**: Dividing a large problem into smaller sub-problems solved simultaneously using multiple processors.

- **Distributed Computing**: Multiple computers working together over a network to solve a problem. Each computer performs part of the computation and results are combined to get the final solution.

- **Supercomputers**: The most powerful type of HPC systems, performing billions of calculations per second.
  Ex: used for weather forecasting, space exploration

Real-World Applications of HPC



Parallel Programming with OpenMP

Outline

- OpenMP Introduction
- Parallel Programming with OpenMP
  - OpenMP parallel region, and worksharing
  - OpenMP data environment, tasking and synchronization
- OpenMP Performance and Best Practices
- More Case Studies and Examples
- Reference Materials

OpenMP

- OpenMP stands for "Open Multi-Processing"
- Standard API to write shared memory parallel applications in C, C++, and Fortran.
- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than T concurrently-executing threads.
  - Hide stack management

Processes and Threads in OpenMP

Process	Thread
• Independent unit of execution that runs on an OS.	• Smaller unit of execution that runs within a process.
• Each process has its own memory space and resources.	• Threads within same process share same memory space
• Processes do not share memory with each other. Communication between processes requires inter-process communication mechanisms.	• Can communicate directly through shared memory, making communication faster and simpler.

OpenMP

- It has three components:

1. Compiler directives
2. Runtime Library routines
3. Environment variables

```
#pragma omp parallel
{
    // Code inside this block runs in parallel
}
```

```
int main() {
    omp_set_num_threads(4); // Set number of threads to 4
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("Thread %d\n", thread_id);
    }
    return 0;
}
```

```
export OMP_NUM_THREADS=4
./your_openmp_program
```

#pragma omp <directive> [clause [[,] clause] …]

“Hello Word” Example/1

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

int main(int argc, char *argv[]) {



          printf("Hello World\n");



   return(0);
}
```

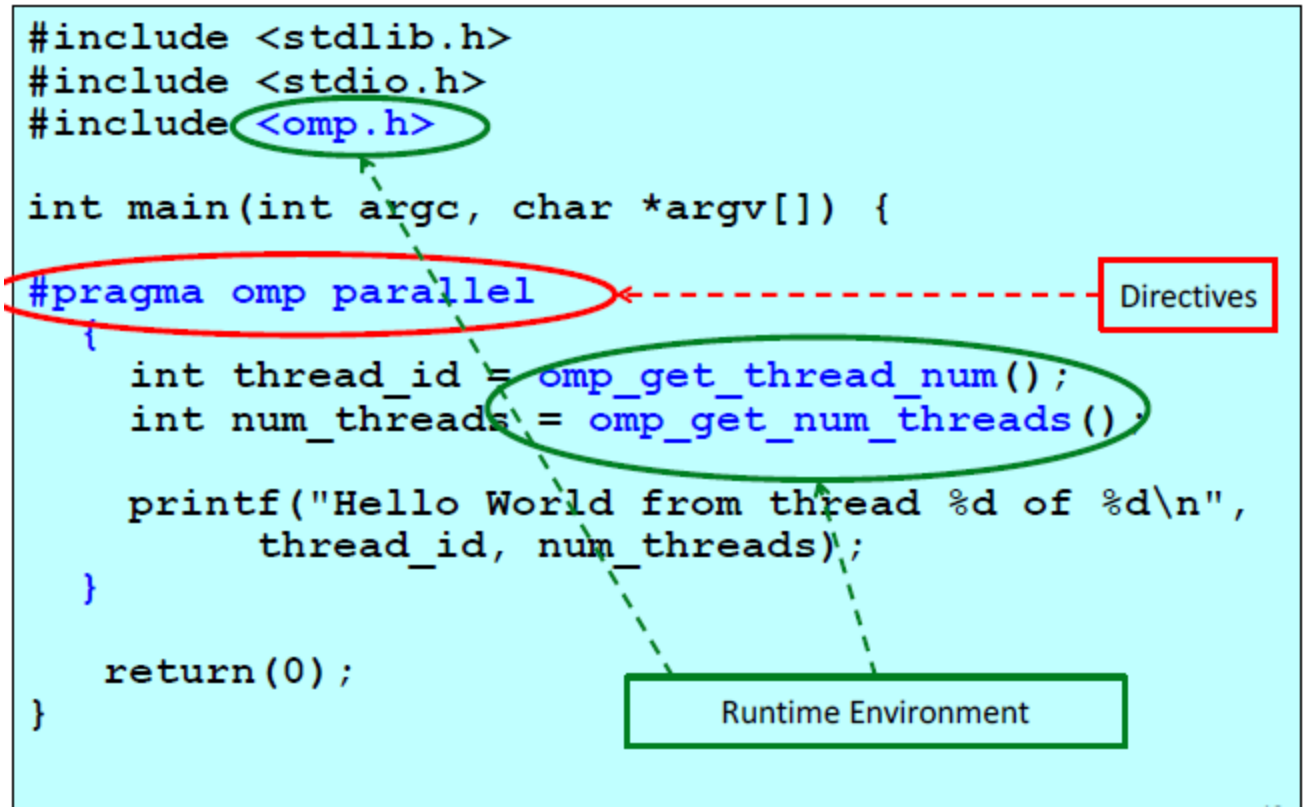
“Hello Word” - An Example/2

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

int main(int argc, char *argv[]) {

    #pragma omp parallel
    {
            printf("Hello World\n");

    } // End of parallel region

    return(0);
}
```

```
$ gcc -fopenmp hello.c


$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World

$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
$
```

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

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("Hello World\n");

   } // End of parallel region

   return(0);
}
```

“Hello Word” - An Example/3



```
$
$ gcc -fopenmp helloomp.c -o helloomp
$ ls helloomp
helloomp
$
$ export OMP_NUM_THREADS=2
$  ./helloomp
Hello World from thread 1 of 2
Hello World from thread 0 of 2
$
$ export OMP_NUM_THREADS=4
$ ./helloomp
Hello World from thread 0 of 4
Hello World from thread 1 of 4
Hello World from thread 3 of 4
Hello World from thread 2 of 4
$
$ export OMP_NUM_THREADS=4
$ ./helloomp
Hello World from thread 1 of 4
Hello World from thread 2 of 4
Hello World from thread 3 of 4
Hello World from thread 0 of 4
```

```
#pragma omp parallel
{
  int thread_id = omp_get_thread_num();
  int num_threads = omp_get_num_threads();

  printf("Hello World from thread %d of %d\n"
         thread_id, num_threads);
}
```

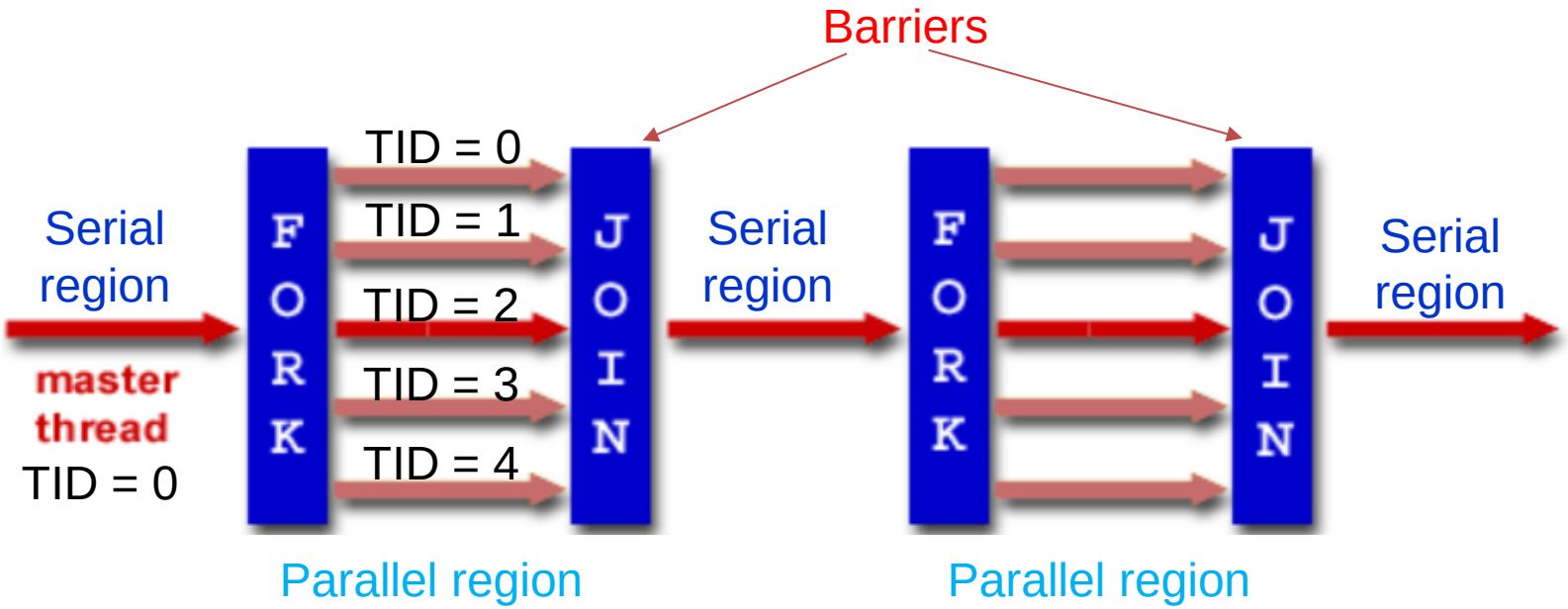
Environment Variable

Environment Variable: it is similar to program arguments used to change the configuration of the execution without recompile the program.

NOTE: the order of print

12

The OpenMP Execution Model (Fork-Join Model)



- **Fork**: The point where a single thread (master thread) splits into multiple parallel threads.
- **Join**: The point where multiple parallel threads complete their tasks and rejoin the master thread.
- A barrier synchronizes all threads at the end of each parallel region.
- After the parallel region exits, the master thread continues, until the next parallel region is encountered.

OpenMP: Controlling Number of Threads

- The number of threads can be set at different levels: program level, pragma level, or by querying and setting the number of available threads and cores.
- At the program level, via the omp_set_number_threads function:

void omp_set_num_threads(int n)

- At the pragma level, via the num_threads clause:

#pragma omp parallel num_threads(numThreads)

```c
int main() {
    // Set the number of threads to 4
    omp_set_num_threads(4);

    #pragma omp parallel
    {
        printf("Hello from thread %d\n", omp_get_thread_num());
    }

    return 0;
}
```

```c
int main() {
    #pragma omp parallel num_threads(2)
    {
        printf("Hello from thread %d\n", omp_get_thread_num());
    }

    return 0;
}
```

OpenMP: Controlling Number of Threads

- Asking how many cores this program has access to:

num = omp_get_num_procs( );

- Setting the number of available threads to the exact number of cores available:

omp_set_num_threads(omp_get_num_procs( ));

- Asking how many OpenMP threads this program is using right now:

num = omp_get_num_threads( );

- Asking which thread number this one is:

me = omp_get_thread_num( )

OpenMP: Controlling Number of Threads

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

int main() {
    // Setting the number of threads to the number of available cores
    omp_set_num_threads(omp_get_num_procs());

    // Parallel region with the set number of threads
    #pragma omp parallel
    {
        int num_threads = omp_get_num_threads();  // Number of threads in the parallel region
        int thread_id = omp_get_thread_num();     // Thread number within the parallel region

        // Only one thread will execute this block and print the number of threads
        #pragma omp single
        {
            printf("Number of threads: %d\n", num_threads);
        }

        // Print a message from each thread
        printf("Hello from thread %d\n", thread_id);
    }

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ vi thread_control.c
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ gcc -fopenmp thread_control.c -o thread_control
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./thread_control
Number of threads: 12
Hello from thread 1
Hello from thread 5
Hello from thread 10
Hello from thread 8
Hello from thread 6
Hello from thread 2
Hello from thread 3
Hello from thread 4
Hello from thread 9
Hello from thread 0
Hello from thread 11
Hello from thread 7
```

Understanding Limitations of OpenMP

- OpenMP **will not**:
  - Parallelize automatically: Explicitly use OpenMP directives to indicate which parts of your code should be parallelized.
  - Guarantee speedup: overhead, load balance
  - Check for data dependencies
  - Provide freedom from data races

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

int main() {
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i < 100; i++) {
        sum += i;  // Potential data race
    }

    printf("Sum: %d\n", sum);
    return 0;
}
```

Different threads might read & write to ‘sum’ at the same time, resulting in an incorrect final value.

  - OpenMP is not overhead-free: Barriers can cause delay.
  - Guarantee identical behavior across vendors or hardware, or even between multiple runs on the same vendor’s hardware
  - Guarantee the order in which threads execute, just that they do execute.

OpenMP: parallel loops

- OpenMP parallel **for** loop: Distribute the iterations of a loop across multiple threads.

```
#pragma omp parallel for [clause [[,] clause] …]
for (initialization; condition; increment) {
    // Loop body
}
```

- To parallelize a loop with OpenMP, it must be in a canonical form:
  - ✓ The loop variable must be an (unsigned) integer or a pointer.
  - ✓ The loop variable should not be modified inside the loop.
  - ✓ The loop condition must be a simple relational expression.
  - ✓ The loop increment must be a constant additive expression.
  - ✓ The number of iterations must be known before the start of the outermost for loop.

OpenMP: parallel loops

- Example of a canonical loop:

```
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
    // Loop body
}
```

- Here, the loop variable *i* is an integer, it is incremented by a constant value, and the condition is a simple relational expression.

Parallelizing Loops with Independent Iterations

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

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  #pragma omp parallel for
    for (int i = 1; i <= max; i++)
      printf ("%d: %d\n", omp_get_thread_num (), i);
  return 0;
}
```

Listing 3.3 Printing out all integers from 1 to *max* in no particular order.

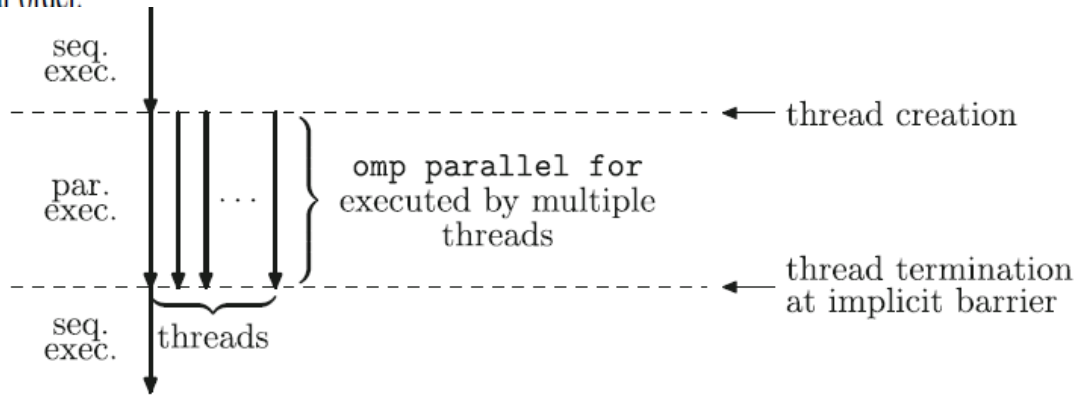


Fig. 3.5 Execution of the program for printing out integers as implemented in Listing 3.3

OpenMP: parallel loops

- The loop variable is made private to each thread in the team.

OpenMP imposes some constraints on loop variable:

- The start and terminate conditions must have compatible types.
- Neither the start nor the terminate conditions can be changed during the execution of the loop.
- The index can only be modified by the increment expression (i.e., not modified inside the loop itself).
- You cannot use a break or a goto to get out of the loop.
- There can be no inter-loop data dependencies such as:

```
A[i]=a[i-1]+1;
```

OpenMP: parallel loops

```
x[ 0 ] = 0.;
y[ 0 ] *= 2.;
for( int i = 1; i < N; i++ )
{
        x[ i ] = x[ i-1 ] + 1.;
        y[ i ] *= 2.;
}
```

Because of the loop dependency, this whole thing is not parallelizable

```
x[ 0 ] = 0.;
for( int i = 1; i < N; i++ )
{
        x[ i ] = x[ i-1 ] + 1.;
}

#pragma omp parallel for shared(y)
for( int i = 0; i < N; i++ )
{
        y[ i ] *= 2.;
}
```

But it can be broken into one loop that is not parallelizable, plus one that is.

OpenMP: parallel loops

Clauses in OpenMP Parallel For Loops:

- Collapse(integer): Specifies how many outermost loops to parallelize together.
- nowait: Eliminates the implicit barrier and thus synchronization at the end of for-loops.

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
        // Loop body
    }
}
```

```
#pragma omp parallel for nowait
for (int i = 0; i < 100; i++) {
    // Loop body
}
```

Without nowait clause

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

int main() {
    int n = 2, m = 2;
    int array[2][2];

    // Initialize array elements in parallel with collapse(2)
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j] = i + j;
            printf("Thread %d initializes array[%d][%d] = %d\n", omp_get_thread_num(), i, j, array[i][j]);
        }
    }

    printf("Initialization done.\n");

    // Process array elements in parallel with collapse(2)
    #pragma omp parallel for collapse(2)
    for (
        f
        }
    }

    print

    retur
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ vi collapse_nowait.c
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ gcc -fopenmp collapse_nowait.c -o collapse_nowait
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./collapse_nowait
Thread 3 initializes array[1][1] = 2
Thread 2 initializes array[1][0] = 1
Thread 1 initializes array[0][1] = 1
Thread 0 initializes array[0][0] = 0
Initialization done.
Thread 0 processes array[0][0] = 5
Thread 1 processes array[0][1] = 6
Thread 2 processes array[1][0] = 6
Thread 3 processes array[1][1] = 7
Processing done.
```

With nowait clause

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

int main() {
    int n = 2, m = 2;
    int array[2][2];

    #pragma omp parallel
    {
        // Initialize array elements in parallel with collapse(2) and nowait
        #pragma omp for collapse(2) nowait
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                array[i][j] = i + j;
                printf("Thread %d initializes array[%d][%d] = %d\n", omp_get_thread_num(), i, j, array[i][j]);
            }
        }

        // Process array elements in parallel with collapse(2) and nowait
        #pragma omp for collapse(2) nowait
        for (int i = 0; i < n; i++) {
    }
    retu
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ vi collapse_nowait1.c
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ gcc -fopenmp collapse_nowait1.c -o collapse_nowait1
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./collapse_nowait1
Thread 0 initializes array[0][0] = 0
Thread 3 initializes array[1][1] = 2
Thread 3 processes array[1][1] = 7
Thread 0 processes array[0][0] = 5
Thread 1 initializes array[0][1] = 1
Thread 1 processes array[0][1] = 6
Thread 2 initializes array[1][0] = 1
Thread 2 processes array[1][0] = 6
```

Single Program Multiple Data (SPMD) in OpenMP

- Parallel programming model where multiple processors execute the same program but operate on different pieces of data.

- Each thread in the parallel region runs the same code but works on different data.

Single Program Multiple Data (SPMD) in OpenMP

Example: add two arrays of integers in parallel using OpenMP. Each thread will operate on a different segment of the arrays.

```c
#define SIZE 4  // Define the size of the array

int main() {
    int A[SIZE], B[SIZE], C[SIZE];

    // Initialize arrays A and B
    for (int i = 0; i < SIZE; i++) {
        A[i] = i;
        B[i] = i * 2;
    }

    // Set the number of threads
    omp_set_num_threads(2);

    // Parallel region with OpenMP
    #pragma omp parallel
    {
        // Each thread will work on a different part of the array
        #pragma omp for
        for (int i = 0; i < SIZE; i++) {
            int thread_id = omp_get_thread_num();  // Get the thread ID
            C[i] = A[i] + B[i];
            printf("Thread %d processing iteration %d: A[%d] = %d, B[%d] = %d, C[%d] = %d\n",
                   thread_id, i, i, A[i], i, B[i], i, C[i]);
        }
    }

    // Print the result array
    printf("\nArray C:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", C[i]);
    }
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ vi SPMD.c
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ gcc -fopenmp SPMD.c -o SPMD
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./SPMD
Thread 0 processing iteration 0: A[0] = 0, B[0] = 0, C[0] = 0
Thread 0 processing iteration 1: A[1] = 1, B[1] = 2, C[1] = 3
Thread 1 processing iteration 2: A[2] = 2, B[2] = 4, C[2] = 6
Thread 1 processing iteration 3: A[3] = 3, B[3] = 6, C[3] = 9

Array C:
0 3 6 9
```

OpenMP: parallel loops

Divide for-loop for parallel sections

```
for (int i=0; i<8; i++) x[i]=0; //run on 4 threads
```

```
#pragma omp parallel
        {                                   // Assume number of threads=4
        int numt=omp_get_num_thread();
        int id = omp_get_thread_num(); //id=0, 1, 2, or 3
        for (int i=id; i<8; i +=numt)
                                x[i]=0;

        }
```

Thread 0	Thread 1	Thread 2	Thread 3
Id=0; x[0]=0; X[4]=0;	Id=1; x[1]=0; X[5]=0;	Id=2; x[2]=0; X[6]=0;	Id=3; x[3]=0; X[7]=0;

- Assume no. of threads = 4.
- Iterations of the loop are divided among 4 threads.
- Each thread will execute a subset of the iterations.

- **Thread 0:**
  i = 0 (initially)
  i = 0 + 4 = 4
- **Thread 1:**
  i = 1 (initially)
  i = 1 + 4 = 5
- **Thread 2:**
  i = 2 (initially)
  i = 2 + 4 = 6
- **Thread 3:**
  i = 3 (initially)
  i = 3 + 4 = 7

Thread 0: Executes iterations 0 & 4
Thread 1: Executes iterations 1 & 5
Thread 2: Executes iterations 2 & 6
Thread 3: Executes iterations 3 & 7

OpenMP: Data sharing

- Various data sharing clauses can be used in omp parallel directive to specify whether and how data are shared among threads:
  - **shared(list)** specifies that each variable in the list is shared by all threads in a team, i.e., all threads share the same copy of the variable. Changes made by one thread are visible to all other threads.
  - **private(list)** specifies that each variable in the list is private to each thread in a team, i.e., each thread has its own local copy of the variable. Changes made by one thread are not visible to other threads.
  - **firstprivate(list)** is like private but each variable listed is initialized with the value it contained when the parallel region was encountered.
  - **lastprivate(list)** is like private but when the parallel region ends each variable listed is updated with its final value within the parallel region.

OpenMP: Data sharing

Example: shared clause

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

int main() {
    int a = 5;

    #pragma omp parallel shared(a)
    {
        printf("Thread %d, a = %d\n", omp_get_thread_num(), a);
        a += omp_get_thread_num();
    }

    printf("Outside parallel region, a = %d\n", a);
    return 0;
}
```

- No. of threads = 4.
- All threads share the same variable ***a*** initialized to 5.
- Each thread prints the initial value of ***a*** and then modifies it.
- The final value of ***a*** outside the parallel region is the result of adding the thread numbers (0 + 1 + 2 + 3 = 6) to the initial value.

```
Thread 0, a = 5
Thread 1, a = 5
Thread 2, a = 5
Thread 3, a = 5
Outside parallel region, a = 11
```

OpenMP: Data sharing

Example: private clause

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

int main() {
    int a = 5;

    #pragma omp parallel private(a)
    {
        a = omp_get_thread_num() * 2;
        printf("Thread %d, private a = %d\n", omp_get_thread_num(), a);
    }

    printf("Outside parallel region, thread ID = %d, a = %d\n", omp_get_thread_num(), a);
    return 0;
}
```

- Each thread gets its own private copy of ***a***.
- Each thread sets its private ***a*** to its thread number multiplied by 2.
- The original value of ***a*** outside the parallel region remains unchanged.

```
Thread 0, private a = 0
Thread 1, private a = 2
Thread 2, private a = 4
Thread 3, private a = 6
Outside parallel region, thread ID = 0, a = 5
```

OpenMP: Data sharing

Example: firstprivate clause

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

int main() {
    int a = 5;

    #pragma omp parallel firstprivate(a)
    {
        a += omp_get_thread_num();
        printf("Thread %d, a = %d\n", omp_get_thread_num(), a);
    }

    printf("Outside parallel region, a = %d\n", a);
    return 0;
}
```

- Each thread gets its own copy of ***a*** initialized to 5.
- Each thread modifies its copy of ***a*** by adding its thread number.
- The original value of ***a*** outside the parallel region remains unchanged.

```
Thread 0, a = 5
Thread 1, a = 6
Thread 2, a = 7
Thread 3, a = 8
Outside parallel region, a = 5
```

OpenMP: Data sharing

Example: lastprivate clause

```
int main() {
    int a = 5;

    #pragma omp parallel for lastprivate(a)
    for (int i = 0; i < 4; i++) {
        a = i;
        printf("Thread %d, iteration %d, a = %d\n", omp_get_thread_num(), i, a);
    }

    printf("Outside parallel region, a = %d\n", a);
    return 0;
}
```

- Each thread gets its own copy of ***a*** during the iterations of the loop.
- Each iteration assigns a value to ***a*** based on the loop index.
- The final value of ***a*** outside the parallel region is updated to the value of *a* from the last iteration (i.e., a = 3).

```
Thread 0, iteration 0, a = 0
Thread 1, iteration 1, a = 1
Thread 2, iteration 2, a = 2
Thread 3, iteration 3, a = 3
Outside parallel region, a = 3
```

OpenMP: Data sharing

Example: shared clause

```c
#include <stdio.h>
#include <omp.h>
int main(){
        int a = 5;
#pragma omp parallel shared(a)
        {
                int b=10;
                a = a+b;
                printf("Thread %d: a = %d\n",omp_get_thread_num(),a);
        }
        printf("Outside parallel region, a = %d\n",a);
        return 0;
}
```

- Thread 0 reads a (initially 5), calculates a + b (5 + 10 = 15), and writes 15 to a.
- Thread 1 reads a (could be 5 or 15 depending on timing), calculates a + b (5 + 10 = 15 or 15 + 10 = 25), and writes the result to a.
- Thread 2 reads a (could be 5, 15, or 25 depending on timing), calculates a + b, and writes the result to a.
- Thread 3 reads a, calculates a + b, and writes the result to a.

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./shared_clause
Thread 0: a = 35
Thread 1: a = 35
Thread 2: a = 35
Thread 3: a = 45
Outside parallel region, a = 45
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./shared_clause
Thread 0: a = 15
Thread 2: a = 25
Thread 1: a = 25
Thread 3: a = 35
Outside parallel region, a = 35
```

OpenMP: Data sharing

Example: private clause

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

int main()
{
        int a=5;
#pragma omp parallel private(a)
        {
         int b=10;
         a=a+b;
         printf("%d\n",a);
        }
        printf("outside:  %d\n",a);
        return 0;
}
```

```
sandhya@telnet:~/PP$ gcc -fopenmp ex1.c
sandhya@telnet:~/PP$ ./a.out
10
10
10
10
outside:  5
sandhya@telnet:~/PP$ ./a.out
10
10
10
10
outside:  5
```

```
32597
10
10
10
Outside: 5
```

- Each private copy of ***a*** is uninitialized because the private clause does not initialize the private copy with the value of the original variable.
- The behavior can be undefined unless OpenMP initializes it to the original a.
- Uninitialized local variables may contain garbage values, leading to unexpected results when used in operations.

OpenMP: Data sharing

Example: firstprivate clause

```c
#include<stdio.h>
#include<omp.h>
int main()
{
        int a=5;
#pragma omp parallel firstprivate(a)
        {
         int b=10;
         a=a+b;
         printf("%d\n",a);
        }
        printf("outside:  %d\n",a);
        return 0;
}
```

```
sandhya@telnet:~/PP$ gcc -fopenmp ex1.c
sandhya@telnet:~/PP$ ./a.out
15
15
15
15
outside:  5
sandhya@telnet:~/PP$ ./a.out
15
15
15
15
outside:  5
sandhya@telnet:~/PP$
```

- To avoid the use of uninitialized private variables, you can use the firstprivate clause instead of private, which initializes each private copy with the original variable's value.

OpenMP: Data sharing

Example: lastprivate clause

```
#include<stdio.h>
#include<omp.h>
int main()
{
        int a=5;
#pragma omp parallel lastprivate(a)
        {
         int b=10;
         a=a+b;
         printf("%d\n",a);
        }
        printf("outside:  %d\n",a);
        return 0;
}
```

```
15
15
15
15
outside: 15
```

OpenMP: Data sharing

- If not specified otherwise,
  - Variables declared outside a parallel construct are shared by default.
  - Variables declared within a parallel construct are private.

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

int main()
{
    int a = 5;
#pragma omp parallel num_threads(5)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region: %d\n", a);
    }
    printf("Outside parallel region: %d\n", a);
    return 0;
}
```

```
Parallel region: 15
Parallel region: 35
Parallel region: 45
Parallel region: 55
Parallel region: 25
Outside parallel region: 55
```

OpenMP: Data sharing

```c
#include <stdio.h>
#include <omp.h>
int main()
{
        int i=0;
        int a=0;
#pragma omp parallel for private(i) lastprivate(a)
        for(i=0;i<5;i++)
        {
                a=i+1;
                printf("Thread %d has a value of a = %d for i = %d\n", omp_get_thread_num(),a,i);
        }
        printf("Value of a after parallel for: a = %d\n",a);
        return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=3
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./private_lastprivate
Thread 0 has a value of a = 1 for i = 0
Thread 2 has a value of a = 5 for i = 4
Thread 0 has a value of a = 2 for i = 1
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
Value of a after parallel for: a = 5
```

- **private(i):** Ensures that each thread has its own private copy of the loop variable *i*.
- **lastprivate(a):** Ensures that after the loop ends, the variable *a* is set to the value it had in the last iteration of the loop executed by any thread.

OpenMP: Data sharing

Default clause:

- The default clause is used to assign a default data-sharing attribute to variables within a parallel region. There are three main forms of the default clause:

1. default(shared): All variables referenced in the parallel region are shared among all threads by default. This can be useful for ensuring that all threads have access to the same data.
2. default(private): All variables in the parallel region are private by default. Each thread has its own local copy of each variable.
   - Not supported in C/C++ (supported only in Fortran)
3. default(none) forces the programmer to specify the data-sharing attribute for each variable explicitly. This is useful for avoiding unintended sharing or privatization of variables and makes the code more readable and less prone to errors.

OpenMP: Data sharing

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

int main() {
    int a = 5;

    #pragma omp parallel default(shared)
    {
        a = a + omp_get_thread_num();
        printf("Thread %d: a = %d\n", omp_get_thread_num(), a);
    }

    printf("Outside: a = %d\n", a);
    return 0;
}
```

```
Thread 0: a = 5
Thread 1: a = 6
Thread 2: a = 7
Thread 3: a = 8
Outside: a = 8
```

OpenMP: Data sharing

```
int main()
{
    int a = 5;
    #pragma omp parallel num_threads(5) default(none) shared(a)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region:a= %d \n", a);
    }

    printf("Outside parallel region: %d\n", a);
    return 0;
}
```

```
Parallel region:a= 15
Parallel region:a= 25
Parallel region:a= 45
Parallel region:a= 35
Parallel region:a= 55
Outside parallel region: 55

D:\DSCA\Parallel_Computing\Lab_Programs\Example_Data
ith code 0.
To automatically close the console when debugging st
le when debugging stops.
Press any key to close this window . . .
```

OpenMP: Data sharing

```c
int main() {
    int a = 5;
    int b = 10;

    #pragma omp parallel default(none) shared(a) private(b)
    {
        b = a + omp_get_thread_num();
        printf("Thread %d: a = %d, b = %d\n", omp_get_thread_num(), a, b);
    }

    printf("Outside: a = %d, b = %d\n", a, b);
    return 0;
}
```

```
Thread 0: a = 5, b = 5
Thread 1: a = 5, b = 6
Thread 2: a = 5, b = 7
Thread 3: a = 5, b = 8
Outside: a = 5, b = 10
```

OpenMP: Data sharing

Nowait clause:

- The nowait clause allows programmers to fine-tune the performance of their parallel programs by controlling synchronization points.
- It is used to suppress the implicit barrier at the end of certain constructs, such as *for*, *sections*, and *single*. - helps to optimize performance by avoiding unnecessary synchronization points.
- **Implicit Barrier:** Normally, OpenMP constructs include an implicit barrier at the end, where all threads must wait until every thread has finished before proceeding.
- Nowait Clause: When the nowait clause is added, this barrier is removed, allowing threads to continue execution without waiting for others to complete.
- "Note: however, that the barrier at the end of a parallel region cannot be suppressed
- One can try to identify places where a barrier is not needed and insert the nowait clause

Example: Nowait clause

```c
int main()
{
#pragma omp parallel
    {
    #pragma omp for
        for (int i = 0; i <5; i++)
        {
            printf("first loop i= %d\n", i);
        }

        printf("outside\n");
    }


    return 0;
}
```

```
first loop i= 2
first loop i= 3
first loop i= 0
first loop i= 4
first loop i= 1
outside
outside
outside
outside
outside
outside
outside
outside
outside

D:\DSCA\Parallel_Computing\Lab_Programs\Nowait_trial\x64\Debug\Now
To automatically close the console when debugging stops, enable To
le when debugging stops.
Press any key to close this window . . .
```

```c
int main()
{
#pragma omp parallel
    {
    #pragma omp for nowait
        for (int i = 0; i <5; i++)
        {
            printf("first loop i= %d\n", i);
        }

        printf("outside\n");
    }


    return 0;
}
```

```
first loop i= 1
outside
first loop i= 3
outside
first loop i= 0
outside
first loop i= 2
outside
outside
first loop i= 4
outside
outside
outside

D:\DSCA\Parallel_Computing\Lab_Programs\Nowait_trial\x64\Debug
To automatically close the console when debugging stops, enabl
le when debugging stops.
Press any key to close this window . . .
```

Example: Nowait clause

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

    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i = 0; i < 8; i++)
        {
            printf("Thread %d processes iteration %d\n", omp_get_thread_num(), i);
        }

        // This section will execute without waiting for the for loop to finish
        #pragma omp single
        {
            printf("Thread %d reached here without waiting for the loop to finish\n", omp_get_thread_num());
        }
    }

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./nowait_data
Thread 0 processes iteration 0
Thread 0 processes iteration 1
Thread 0 reached here without waiting for the loop to finish
Thread 3 processes iteration 6
Thread 3 processes iteration 7
Thread 2 processes iteration 4
Thread 2 processes iteration 5
Thread 1 processes iteration 2
Thread 1 processes iteration 3
```

OpenMP: Data sharing

Schedule clause:

- The schedule clause is supported on the loop construct only.
- It is used to control how loop iterations are distributed over the threads, which can have a major impact on the performance of a program.
- The syntax is:
  - **schedule(kind [, chunk_size])**
- **chunk_size parameter specifies the size of each chunk.**
- The granularity of this workload distribution is a chunk.
  - **Chunk: A contiguous, nonempty subset of the iteration space.**

```
#pragma omp for schedule(static, 2)
for (int i = 0; i < 10; i++) {
    // loop body
}
```

OpenMP: Data sharing

Schedule clause:

Types of scheduling strategies that can be used with schedule clause:

1. Static
2. Dynamic
3. Guided
4. Runtime
5. Auto

OpenMP: Data sharing

Schedule clause:

1. Static
   - Iterations are divided into equal sized chunks of size chunk size, and each chunk is assigned to a thread before the loop starts. (iterations are evenly divided among threads.)
   - The chunks are assigned to the threads statically in a **round-robin** manner, in the **order of the thread number.**
   - This approach is most effective when the workload is evenly distributed and predictable.
   - The last chunk to be assigned may have a smaller number of iterations.
   - When no chunk size is specified, the iteration space is divided into chunks that are approximately equal in size.
   - In this approach, iterations are divided equally among the threads before they start working. Each thread knew exactly which iterations they had to complete.

OpenMP: Data sharing

Schedule clause: Static

```
#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);
}
```

```
Thread 0 processes iteration 0
Thread 0 processes iteration 1
Thread 1 processes iteration 2
Thread 1 processes iteration 3
Thread 2 processes iteration 4
Thread 2 processes iteration 5
Thread 3 processes iteration 6
Thread 3 processes iteration 7
Thread 0 processes iteration 8
Thread 0 processes iteration 9
```

- Directive tells the compiler to use static scheduling with chunks of 2. If there are 4 threads, each thread gets 2 consecutive iterations of the loop to work on.
- **Use Case:** Best for loops where each iteration takes roughly the same amount of time.

OpenMP: Data sharing

Schedule clause:

2. Dynamic
   - The iterations are assigned to threads as the threads request them.
   - Chunks of iterations are assigned to threads on the fly as they complete their current chunks.
   - The thread executes the chunk of iterations (controlled through the chunk size parameter), then requests another chunk until there are no more chunks to work on.
   - The last chunk may have fewer iterations than chunk size. When no chunk size is specified, it defaults to 1.
   - In this approach, chunks re assigned to threads as they became available. When a thread finished a task, it receives the next available chunk from a queue.

OpenMP: Data sharing

Schedule clause: Dynamic

```
#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);
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./dynamic_schedule
Thread 3 processes iteration 0
Thread 3 processes iteration 1
Thread 3 processes iteration 8
Thread 3 processes iteration 9
Thread 0 processes iteration 2
Thread 0 processes iteration 3
Thread 1 processes iteration 4
Thread 1 processes iteration 5
Thread 2 processes iteration 6
Thread 2 processes iteration 7
```

- With a chunk size of 2, iterations are assigned to threads in chunks of 2 as threads become available.
- **Use Case:** This is useful when the workload is uneven or unpredictable.

OpenMP: Data sharing

Schedule clause:

3. Guided
   - The iterations are assigned to threads as the threads request them.
   - The chunk size starts large and decreases exponentially.
   - The thread executes the chunk of iterations (controlled through the chunk size parameter), then requests another chunk until there are no more chunks to work on.
   - Both the dynamic and guided schedules are useful for handling poorly balanced and unpredictable workloads.
   - The difference between them is that with the guided schedule, the size of the chunk (of iterations) decreases over time.
   - It combines elements of static and dynamic scheduling. Initially, larger chunks of iteration will be assigned to the threads, but as iteration completes, the chunk size reduces.

OpenMP: Data sharing

Schedule clause: Guided

```
#pragma omp parallel for schedule(guided, 2)
for (int i = 0; i < 10; i++) {
    printf("Thread %d processes iteration %d\n", omp_get_thread_num(), i);
}
```

```
Thread 0 processes iteration 0
Thread 0 processes iteration 1
Thread 1 processes iteration 2
Thread 1 processes iteration 3
Thread 2 processes iteration 4
Thread 2 processes iteration 5
Thread 3 processes iteration 6
Thread 3 processes iteration 7
Thread 0 processes iteration 8
Thread 1 processes iteration 9
```

Use Case: Suitable for loops where the workload decreases over time.

OpenMP: Data sharing

Schedule clause:

4. Runtime

– If this schedule is selected, the decision regarding scheduling kind is made at run time.

– Instead of making a compile time decision, the OpenMP **OMP_SCHEDULE environment variable** can be used to choose the schedule and (optional) *chunk size* at run time.

export OMP_SCHEDULE="dynamic,2"

OpenMP: Data sharing

Schedule clause:

5. Auto

   – The compiler and runtime system decide the best scheduling strategy. This can help leverage system-specific optimizations.

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

OpenMP: Synchronization Constructs

- In parallel computing, multiple threads often need to access shared data.
- Synchronization constructs in OpenMP help manage the interaction between threads to ensure data consistency.
- These constructs are essential when implicit barriers are not sufficient to prevent race conditions or other concurrency issues.

OpenMP: Synchronization Constructs

- Barrier
- Ordered
- Critical
- Atomic
- Locks
- Master

Synchronization Constructs: Barrier

- In parallel computing, synchronization points are essential to ensure that multiple threads execute certain parts of a program in a coordinated manner.
- A **barrier** is one such synchronization construct where threads wait for each other at a specified point before proceeding.
- The compiler automatically inserts an implicit barrier at the end of most parallel constructs.
- **Barriers** are explicitly used when you need to synchronize threads at specific points in the program.

```
#pragma omp barrier
```

Synchronization Constructs: Barrier

- Two important restrictions apply to the barrier construct:
  - Each barrier must be encountered by all threads in a team, or by none at all.
  - The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in the team.

Synchronization Constructs: Barrier

Why Use Barriers:

- Synchronization: Ensures that all threads have completed a specific part of the code before moving on.
- Data Consistency: Ensures that all threads have updated shared data before any thread uses it. This prevents data races and ensures that threads have the correct and most recent data.
- Coordination: Helps in coordinating the work of multiple threads to ensure correct program execution. They control the order of execution in parallel regions, which is critical for tasks that depend on the results of other threads. Without barriers, threads might proceed in an unpredictable manner, leading to incorrect results.

Common Use Cases:

- Data Initialization: Ensuring that all threads complete data initialization before proceeding with computations.
- Phased Computation: In algorithms that proceed in phases, barriers can ensure all threads complete one phase before starting the next.

Synchronization Constructs: Barrier

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

#define N 4

int main() {
    int array[N];
    int i;

    // Parallel region to initialize the array
    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        array[i] = i;
        printf("Thread %d initializes array[%d] = %d\n", omp_get_thread_num(), i, array[i]);
    }

    // Barrier to ensure all threads have finished initialization
    #pragma omp barrier

    // Parallel region to process the array
    #pragma omp parallel for
    for (i = 0;
        array[i]
        printf("
    }

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ vi barrier_exp.c
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ gcc -fopenmp barrier_exp.c -o barrier_exp
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./barrier_exp
Thread 0 initializes array[0] = 0
Thread 3 initializes array[3] = 3
Thread 1 initializes array[1] = 1
Thread 2 initializes array[2] = 2
Thread 3 processes array[3] = 13
Thread 1 processes array[1] = 11
Thread 2 processes array[2] = 12
Thread 0 processes array[0] = 10
```

Synchronization Constructs: Barrier

Example: Each thread will perform two tasks. The first task will be a simple print statement followed by a delay for half of the threads. After this, all threads will wait at a barrier before proceeding to the second task.

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

int main() {
    int TID;

    #pragma omp parallel private(TID)
    {
        TID = omp_get_thread_num();
        printf("Thread %d: Starting

        if (TID < omp_get_num_thread
            // Simulate work with a
            printf("Thread %d: Worki
            system("sleep 5");
        } else {
            // Immediate completion
            printf("Thread %d: Finis
        }

        printf("Thread %d: Reached b

        #pragma omp barrier

        printf("Thread %d: Passed ba
        // Simulate second task
        printf("Thread %d: Completin
    }

    printf("All threads have complet
    return 0;
}
```

```
Thread 0: Starting first task.
Thread 0: Working on first task (sleeping for 5 seconds).
Thread 3: Starting first task.
Thread 3: Finished first task quickly.
Thread 3: Reached barrier, waiting for others...
Thread 1: Starting first task.
Thread 1: Working on first task (sleeping for 5 seconds).
Thread 2: Starting first task.
Thread 2: Finished first task quickly.
Thread 2: Reached barrier, waiting for others...
Thread 0: Reached barrier, waiting for others...
Thread 1: Reached barrier, waiting for others...
Thread 0: Passed barrier, starting second task.
Thread 0: Completing second task.
Thread 1: Passed barrier, starting second task.
Thread 1: Completing second task.
Thread 3: Passed barrier, starting second task.
Thread 3: Completing second task.
Thread 2: Passed barrier, starting second task.
Thread 2: Completing second task.
All threads have completed their tasks.
```

Synchronization Constructs: Ordered

- It is a synchronization mechanism that allows certain blocks of code within a parallel loop to be executed in sequential order.
- Useful when you need to enforce a specific order of execution, such as printing data computed by different threads or ensuring a sequence of operations.

Common Use Cases:

- This can be very useful for debugging purposes, as it helps us to determine whether there are any data races in our code.
- It ensures that the enclosed block of code is executed in the order of the loop iterations.

```
#pragma omp ordered
        structured block
```

Synchronization Constructs: Ordered

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

int main() {
    int n = 10;
    int array[10];

    // Initialize array elements in parallel
    #pragma omp parallel for ordered
    for (int i = 0; i < n; i++) {
        array[i] = i * 2;

        // Enf
        #pragm
        {
            pr
        }
    }

    printf("Ar

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ vi ordered_sync.c
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ gcc -fopenmp ordered_sync.c
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./ordered_sync
Thread 0, array[0] = 0
Thread 0, array[1] = 2
Thread 0, array[2] = 4
Thread 1, array[3] = 6
Thread 1, array[4] = 8
Thread 1, array[5] = 10
Thread 2, array[6] = 12
Thread 2, array[7] = 14
Thread 3, array[8] = 16
Thread 3, array[9] = 18
Array initialization done.
```

Synchronization Constructs: Ordered

```c
#include <stdio.h>
#include <omp.h>
int main()
{
        int n=11;
        int i,sum=0,TID;
#pragma omp parallel default(none) shared(n,sum) private(i,TID)
        {
                TID = omp_get_thread_num();
                int sumLocal = 0;
#pragma omp for ordered
                for(i=0;i<n;i++)
                {
                        sumLocal +=i;
#pragma omp ordered
                        {
                                sum +=sumLocal;
                                printf("TID =%d: sumLocal=%d: sum=%d\n",TID,sumLocal,sum);
                        }
                }
        }
        printf("Outside parallel region\n");
        printf("Value of sum after parallel region:%d\n",
        return 0;
}
```

Without ordered construct

```
TID =0: sumLocal=0: sum=18
TID =0: sumLocal=1: sum=19
TID =0: sumLocal=3: sum=22
TID =1: sumLocal=3: sum=3
TID =1: sumLocal=7: sum=29
TID =1: sumLocal=12: sum=41
TID =3: sumLocal=9: sum=18
TID =3: sumLocal=19: sum=60
TID =2: sumLocal=6: sum=9
TID =2: sumLocal=13: sum=73
TID =2: sumLocal=21: sum=94
Outside parallel region
Value of sum after parallel region:94
```

With ordered construct

```
TID =0: sumLocal=0: sum=0
TID =0: sumLocal=1: sum=1
TID =0: sumLocal=3: sum=4
TID =1: sumLocal=3: sum=7
TID =1: sumLocal=7: sum=14
TID =1: sumLocal=12: sum=26
TID =2: sumLocal=6: sum=32
TID =2: sumLocal=13: sum=45
TID =2: sumLocal=21: sum=66
TID =3: sumLocal=9: sum=75
TID =3: sumLocal=19: sum=94
Outside parallel region
Value of sum after parallel region:94
```

- The #pragma omp ordered directive ensures that the updates to the shared variable *sum* are done in an ordered and sequential manner.

Synchronization Constructs: Ordered

```c
#include <stdio.h>
#include <omp.h>
int main()
{
        int n =8;
        int a[8]={};
        int i, TID;
#pragma omp parallel for default(none) ordered schedule(static) shared(n,a)
        for(i=0;i<n;i++)
        {
                int TID = omp_get_thread_num();
                printf("Thread %d updates a[%d]\n",TID,i);
                a[i]+=i;
#pragma omp ordered
                {
                        printf("Thread %d prints value of a[%d] = %d\n",TID,i,a[i]);
                }
        }
        return 0;
}
```

```
Thread 1 updates a[1]
Thread 4 updates a[4]
Thread 2 updates a[2]
Thread 7 updates a[7]
Thread 0 updates a[0]
Thread 0 prints value of a[0] = 0
Thread 1 prints value of a[1] = 1
Thread 3 updates a[3]
Thread 5 updates a[5]
Thread 6 updates a[6]
Thread 2 prints value of a[2] = 2
Thread 3 prints value of a[3] = 3
Thread 4 prints value of a[4] = 4
Thread 5 prints value of a[5] = 5
Thread 6 prints value of a[6] = 6
Thread 7 prints value of a[7] = 7
```

Synchronization Constructs: Critical

- The critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously.
- The associated code is referred to as a critical region, or a critical section.
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.

Synchronization Constructs: Critical

```c
#include <stdio.h>
#include <omp.h>
int main()
{
        int n=11;
        int i,sum=0,TID;
#pragma omp parallel default(none) shared(n,sum) private(i,TID)
        {
                TID = omp_get_thread_num();
                int sumLocal = 0;
#pragma omp for
                for(i=0;i<n;i++)
                {
                        sumLocal +=i;
#pragma omp critical
                        {
                                sum +=sumLocal;
                                printf("TID =%d: sumLocal=%d: sum=%d\n",TID,sumLocal,sum);
                        }
                }
        }
        printf("Outside parallel
        printf("Value of sum aft
        return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./critical_sync2
TID =0: sumLocal=0: sum=0
TID =0: sumLocal=1: sum=1
TID =0: sumLocal=3: sum=4
TID =1: sumLocal=3: sum=7
TID =1: sumLocal=7: sum=14
TID =1: sumLocal=12: sum=26
TID =2: sumLocal=6: sum=32
TID =3: sumLocal=9: sum=41
TID =3: sumLocal=19: sum=60
TID =2: sumLocal=13: sum=73
TID =2: sumLocal=21: sum=94
Outside parallel region
Value of sum after parallel region:94
```

Synchronization Constructs: Ordered Vs Critical

```
TID =0: sumLocal=0: sum=0
TID =0: sumLocal=1: sum=1
TID =0: sumLocal=3: sum=4
TID =1: sumLocal=3: sum=7
TID =1: sumLocal=7: sum=14
TID =1: sumLocal=12: sum=26
TID =2: sumLocal=6: sum=32
TID =2: sumLocal=13: sum=45
TID =2: sumLocal=21: sum=66
TID =3: sumLocal=9: sum=75
TID =3: sumLocal=19: sum=94
Outside parallel region
Value of sum after parallel region:94
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./critical_sync2
TID =0: sumLocal=0: sum=0
TID =0: sumLocal=1: sum=1
TID =0: sumLocal=3: sum=4
TID =1: sumLocal=3: sum=7
TID =1: sumLocal=7: sum=14
TID =1: sumLocal=12: sum=26
TID =2: sumLocal=6: sum=32
TID =3: sumLocal=9: sum=41
TID =3: sumLocal=19: sum=60
TID =2: sumLocal=13: sum=73
TID =2: sumLocal=21: sum=94
Outside parallel region
Value of sum after parallel region:94
```

Synchronization Constructs: Atomic

- Purpose: The atomic construct allows multiple threads to safely update shared data without interfering with each other.
- Scope: It is applied only to a single assignment statement that immediately follows it.

```
#pragma omp atomic
x += 1;
```

- When a thread reaches an atomic construct, it locks the memory location being updated.
- This lock ensures that no other thread can access or modify that memory location until the current thread finishes its update.
- Once the update is complete, the lock is released, allowing other threads to proceed.
- Efficiency: The atomic construct is more efficient than a critical region because it locks only the specific memory location being updated, rather than a larger block of code.

#pragma omp atomic
statement

Synchronization Constructs: Atomic

Example: Without atomic

```
Thread 0 processing iteration 0: x after increment = 1
Thread 0 processing iteration 1: x after increment = 3
Thread 3 processing iteration 4: x after increment = 4
Thread 1 processing iteration 2: x after increment = 2
Thread 2 processing iteration 3: x after increment = 1
Final value of x: 4
```

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

int main() {
    int x = 0;

    #pragma omp parallel for
    for (int i = 0; i < 5; i++) {
        // Get the thread number
        int thread_num = omp_get_thread_num();
        // Increment x
        x += 1;

        // Print the value of x after increment
        printf("Thread %d processing iteration %d: x after increment = %d\n", thread_num, i, x);
    }

    printf("Final value of x: %d\n", x);
    return 0;
}
```

- Without the atomic construct, race conditions can cause unpredictable and inconsistent results when multiple threads attempt to update a shared variable simultaneously.
- Output could be any value between 1 and 5, depending on how the threads interleave their execution.
- When multiple threads execute x += 1; simultaneously, they might read the same value of x before any of them updates it.
- For example, if x is initially 0, and two threads read x as 0 before either of them writes back their incremented value, they might both write back 1, effectively losing one increment.

```
Thread 1 processing iteration 2: x after increment = 1
Thread 3 processing iteration 4: x after increment = 3
Thread 2 processing iteration 3: x after increment = 2
Thread 0 processing iteration 0: x after increment = 4
Thread 0 processing iteration 1: x after increment = 5
Final value of x: 5
```

Synchronization Constructs: Atomic

Example: With atomic

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

int main() {
    int x = 0;

    #pragma omp parallel for
    for (int i = 0; i < 5; i++) {
        // Get the thread number
        int thread_num = omp_get_thread_num();
#pragma omp atomic
        // Increment x
        x += 1;

        // Print the value of x after increment
        printf("Thread %d processing iteration %d: x after increment = %d\n", thread_num, i, x);
    }

    printf("Final value of x: %d\n", x);
    return 0;
}
```

```
Thread 0 processing iteration 0: x after increment = 1
Thread 1 processing iteration 1: x after increment = 2
Thread 2 processing iteration 2: x after increment = 3
Thread 3 processing iteration 3: x after increment = 4
Thread 0 processing iteration 4: x after increment = 5
Final value of x: 5
```

Synchronization Constructs: Locks

- Locks in OpenMP provide a mechanism to ensure that only one thread can access a critical section of code at a time.
- They are used to avoid race conditions and ensure data integrity in parallel programming.

States of an OpenMP Locks

1. Uninitialized: The lock has been declared but not yet initialized.
2. Unlocked: The lock is initialized and currently not held by any thread.
3. Locked: The lock is held by a thread, and no other thread can acquire it until it is released.

Synchronization Constructs: Locks

Workflow of Using Locks

- Declare: Define the lock variable using omp_lock_t.
- Initialize: Set up the lock with omp_init_lock(). This changes its state from uninitialized to unlocked.
- Set (Lock): Use omp_set_lock() to acquire the lock. If the lock is already held by another thread, the thread calling omp_set_lock() will block until the lock becomes available.
- Use omp_test_lock() to attempt to acquire the lock without blocking. If the lock is not available, it returns a non-zero value, allowing the thread to continue executing.
- Unset (Unlock): Release the lock with omp_unset_lock(). This changes its state from locked to unlocked, allowing other threads to acquire it.
- Destroy: Clean up the lock with omp_destroy_lock() to deallocate any resources associated with the lock.

Synchronization Constructs: Locks

```c
#include <stdio.h>

int main() {
    int x = 0;  // Shared variable
    omp_lock_t lock;  // Declare lock

    omp_init_lock(&lock);  // Initialize lock

    #pragma omp parallel for
    for (int i = 0; i < 5; i++) {
        int thread_num = omp_get_thread_num();  // Get the thread number

        omp_set_lock(&lock);  // Set the lock (block if already locked)

        x += 1;  // Increment x
        printf("Thread %d processing iteration %d: x after increment = %d\n", thread_num, i, x);

        omp_unset_lock(&lock);  // Release the lock
    }

    omp_destroy_lock(&lock);  // Destroy lock

    printf("Final value of x: %d\n", x);  // Print final value
    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~$ ./lock_sync
Thread 0 processing iteration 0: x after increment = 1
Thread 1 processing iteration 2: x after increment = 2
Thread 2 processing iteration 3: x after increment = 3
Thread 3 processing iteration 4: x after increment = 4
Thread 0 processing iteration 1: x after increment = 5
Final value of x: 5
```

Synchronization Constructs: Locks

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

omp_lock_t simple_lock;

int main() {
    omp_init_lock(&simple_lock);

    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();

        while (!omp_test_lock(&simple_lock))
            printf("Thread %d - failed to acquire simple_lock\
                    tid);

        printf("Thread %d - acquired simple_lock\n", tid);

        printf("Thread %d - released simple_lock\n", tid);
        omp_unset_lock(&simple_lock);
    }

    omp_destroy_lock(&simple_lock);
}
```

```
Thread 0 - failed to acquire simple_lock
Thread 3 - acquired simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 0 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - failed to acquire simple_lock
Thread 3 - released simple_lock
Thread 0 - failed to acquire simple_lock
Thread 0 - acquired simple_lock
Thread 0 - released simple_lock
Thread 2 - failed to acquire simple_lock
Thread 2 - acquired simple_lock
Thread 2 - released simple_lock
Thread 1 - failed to acquire simple_lock
Thread 1 - acquired simple_lock
Thread 1 - released simple_lock
```

Synchronization Constructs: Master

Single Thread Execution:

- The master construct ensures that the enclosed block of code is executed by only the master thread (usually thread 0 in a parallel region).
- This is similar to the single construct but with some important differences.

No Implicit Barrier:

- Unlike the single construct, the master construct does not come with an implicit barrier at the end.
- This means that other threads are not automatically synchronized with the master thread after the master block.
- This lack of an implicit barrier means you must explicitly manage synchronization if needed.

Usage:

- It is typically used for operations that need to be performed once, such as initializing data or printing messages, and should be performed by only one thread to avoid redundancy.

Synchronization Constructs: Master

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

int main() {
    int data = 0; // Shared variable

    // Initialize OpenMP parallel region
    #pragma omp parallel
    {
        // Only the master thread will execute this block
        #pragma omp master
        {
            // Initialization performed only once by the master thread
            data = 10;
            printf("Master thread %d initialized data to %d\n",omp_get_thread_num(), data);
        }

        // All threads execute this block
        #pragma omp for
        for (int i = 0; i < 5; i++) {
            printf("Thread %d processing iteratic
                    omp_get_thread_num(), i, data)
        }
    }

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~$ export OMP_NUM_THREADS=5
manisha@MIT-DSCA-FL2158:~$ ./master_sync
Master thread 0 initialized data to 10
Thread 0 processing iteration 0 with data = 10
Thread 4 processing iteration 4 with data = 10
Thread 1 processing iteration 1 with data = 10
Thread 3 processing iteration 3 with data = 0
Thread 2 processing iteration 2 with data = 10
manisha@MIT-DSCA-FL2158:~$ ./master_sync
Thread 4 processing iteration 4 with data = 0
Master thread 0 initialized data to 10
Thread 0 processing iteration 0 with data = 10
Thread 3 processing iteration 3 with data = 10
Thread 1 processing iteration 1 with data = 10
Thread 2 processing iteration 2 with data = 0
```

OpenMP: The Sections Construct

- The sections construct in OpenMP allows to specify different sections of code to be executed by different threads.
- Useful when you have distinct tasks that can be performed in parallel but are independent of each other.

The construct consists of two directives:

1. **#pragma omp sections:** This directive specifies that the enclosed blocks of code are sections that should be executed by different threads.
2. **#pragma omp section:** This directive marks the beginning of a new section within the sections construct.

OpenMP: The Sections Construct

- Each section must be a structured block of code that is independent of the other sections. This means that one section should not depend on the data or the results of another section.
- At runtime, the code blocks specified by the section directives are executed by the threads in the team. Each thread will execute one code block at a time, and each code block will be executed exactly once.

```
#pragma omp sections [clause[[,] clause]...]
    {
    [#pragma omp section ]
        structured block
    [#pragma omp section
        structured block ]
    ...
    }
```

OpenMP: The Sections Construct

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

// Task functions
void task1() {
    printf("Task 1 executed by thread %d\n", omp_get_thread_num());
}

void task2() {
    printf("Task 2 executed by thread %d\n", omp_get_thread_num());
}

void task3() {
    printf("Task 3 executed by thread %d\n", omp_get_thread_num());
}

int main() {
    // Parallel region with sections
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                task1(); // Task 1
            }

            #pragma omp section
            {
                task2(); // Task 2
            }

            #pragma omp section
            {
                task3(); // Task 3
            }
        } // end of sections
    } // end of parallel region

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=3
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./section_construct
Task 2 executed by thread 2
Task 1 executed by thread 0
Task 3 executed by thread 1
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./section_construct
Task 1 executed by thread 0
Task 2 executed by thread 1
Task 3 executed by thread 2
```

OpenMP: The Sections Construct

```c
#include <stdio.h>
#include <omp.h>
int main(){
        int a =1, b=1, c=2, d=2;
#pragma omp parallel
        {
#pragma omp sections
                {
#pragma omp section
                        (void) funcA(a,b, omp_get_thread_num());
#pragma omp section
                        (void) funcB(c,d, omp_get_thread_num());
                }
        }
        return 0;
}
void funcA(int a, int b, int TID)
{
        printf("Work done by thread %d is: sum = %d\n",TID,a+b);
}
void funcB(int c, int d, int TID)
{
        printf("Work done by thread %d is: sum = %d\n",TID,c+d);
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=2
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./section_construct1
Work done by thread 1 is: sum = 4
Work done by thread 0 is: sum = 2
```

OpenMP: Single Construct

- The **single** construct in OpenMP is used to specify that a block of code should be executed by only one thread in a team.
- This is useful when there is some initialization or a critical section that only needs to be executed once, irrespective of the number of threads.
- **#pragma omp single -** This directive specifies that the following block of code should be executed by only one thread.

#pragma omp single *[clause[[,] clause]...]* *structured block*

OpenMP: Single Construct

- **No Specific Thread:** OpenMP does not specify which thread will execute the single block; it is up to the runtime to decide.
- **Implicit Barrier:** By default, there is an implicit barrier at the end of the single construct. This means all other threads in the team will wait until the single thread completes the execution of the code block.
- If you do not want the implicit barrier, you can use the **nowait** clause to remove it.

OpenMP: Single Construct

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

int main() {
    int n = 10;
    int array[n];

    // Parallel region
    #pragma omp parallel
    {
        // Single construct
        #pragma omp single
        {
            printf("Array initialization by thread %d\n", omp_get_thread_num());
        }

        // Parallel for loop to initialize the array
        #pragma omp for
        for (int i = 0; i < n; i++) {
            array[i] = i * 2;
            printf("Thread %d setting array[%d] to %d\n", omp_get_thread_num(), i, array[i]);
        }
    } // end of parallel region

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./single_construct
Array initialization by thread 2
Thread 0 setting array[0] to 0
Thread 0 setting array[1] to 2
Thread 0 setting array[2] to 4
Thread 1 setting array[3] to 6
Thread 1 setting array[4] to 8
Thread 1 setting array[5] to 10
Thread 3 setting array[8] to 16
Thread 3 setting array[9] to 18
Thread 2 setting array[6] to 12
Thread 2 setting array[7] to 14
```

OpenMP: Single Construct

```
int a=0,b[10];
int i=0;

#pragma omp parallel shared(a,b) private(i)
{
    #pragma omp single
    {
    a = 10;
    printf("Single construct executed by thread %d\n",
    omp_get_thread_num());
    }
    /* A barrier is automatically inserted here */
    #pragma omp for
    for (i=0; i<10; i++)
    b[i] = a;
} /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<10; i++)
printf("b[%d] = %d\n",i,b[i]);
```

```
Single construct executed by thread 0
After the parallel region:
b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
b[5] = 10
b[6] = 10
b[7] = 10
b[8] = 10
b[9] = 10

--------------------------------
Process exited after 0.2178 seconds with retur
Press any key to continue . . .
```

OpenMP: Combined Parallel Work-Sharing Constructs

- In OpenMP, combined parallel work-sharing constructs provide a convenient way to simplify the syntax when a parallel region consists of exactly one work-sharing construct.
- These shortcuts combine the parallel directive with one of the work-sharing constructs (such as for, sections, or single), allowing you to write more concise and readable code.

Combined Constructs

1. **Parallel For:** #pragma omp parallel for
2. **Parallel Sections:** #pragma omp parallel sections
3. **Parallel Single:** #pragma omp parallel single

OpenMP: Combined Parallel Work-Sharing Constructs

Full version	Combined construct

#pragma omp parallel
{
  #pragma omp for
  for-loop
}
	#pragma omp parallel for
	  for-loop

#pragma omp parallel
{
#pragma omp sections
{
  [**#pragma omp section**]
  structured block
  [**#pragma omp section**
  structured block]
  ...
}
}
	#pragma omp parallel sections
	{
	  [**#pragma omp section**]
	  structured block
	  [**#pragma omp section**
	  structured block]
	  ...
	}

Benefits

- **Simplicity:** Reduces the amount of code you need to write.
- **Readability:** Makes the code easier to read and understand.
- **Convenience:** Eliminates the need to explicitly define a separate parallel region.

OpenMP: If Clause

- The **if** clause in OpenMP is used with the parallel construct only, where it is used to control the execution of parallel regions based on a logical condition.
- Since some overheads are inevitably incurred with the creation and termination of a parallel region, it is sometimes necessary to test whether there is enough work in the region to warrant its parallelization.
- #pragma omp parallel if(condition): This directive specifies that the following block of code should be executed in parallel only if the condition evaluates to true.

if(scalar-logical-expression)

- If the logical expression evaluates to true, which means it is of type integer and has a non-zero value in C/C++, the parallel region will be executed by a team of threads
- If it evaluates to false, the region is executed by a single thread only

OpenMP: If Clause

```
int n=5;
int TID=0;
#pragma omp parallel if (n > 5) default(none) \
private(TID) shared(n)
{
    TID = omp_get_thread_num();
    #pragma omp single
    {
        printf("Value of n = %d\n",n);
        printf("Number of threads in parallel region: %d\n",
        omp_get_num_threads());
    }
    printf("Print statement executed by thread %d\n",TID);
} /*-- End of parallel region --*/
```

```
Value of n = 5
Number of threads in parallel region: 1
Print statement executed by thread 0

--------------------------------
Process exited after 0.2263 seconds with return value 0
Press any key to continue . . .
```

```
int n=6;
int TID=0;
#pragma omp parallel if (n > 5) default(none) \
private(TID) shared(n)
{
    TID = omp_get_thread_num();
    #pragma omp single
    {
        printf("Value of n = %d\n",n);
        printf("Number of threads in parallel region: %d\n",
        omp_get_num_threads());
    }
    printf("Print statement executed by thread %d\n",TID);
} /*-- End of parallel region --*/
```

```
Value of n = 6
Number of threads in parallel region: 8
Print statement executed by thread 2
Print statement executed by thread 6
Print statement executed by thread 0
Print statement executed by thread 4
Print statement executed by thread 1
Print statement executed by thread 5
Print statement executed by thread 7
Print statement executed by thread 3

--------------------------------
Process exited after 0.2341 seconds with return value 0
Press any key to continue . . .
```

OpenMP: If Clause

```c
int main() {
    int n = 6;  // Change this value to test the condition
    int array[n];

    // Condition to decide whether to execute in parallel
    int parallel_condition = (n > 5);

    // Parallel region with if clause
    #pragma omp parallel if(parallel_condition)
    {
        #pragma omp for
        for (int i = 0; i < n; i++) {
            array[i] = i * 2;
            printf("Thread %d setting array[%d] to %d\n", omp_get_thread_num(), i, array[i]);
        }

        #pragma omp single
        {
            if (parallel_condition) {
                printf("Executed in parallel by multiple threads.\n");
                printf("Executed by thread %d\n",omp_get_thread_num());
            } else {
                printf("Executed serially by a single thread %d\n",omp_get_thread_num());
            }
        }
    } // end of parallel region

    return 0;
}
```

OpenMP: If Clause

n = 6

```
manisha@MIT-DSCA-FL2158:~$ export OMP_NUM_THREADS=3
manisha@MIT-DSCA-FL2158:~$ ./if_clause
Thread 2 setting array[4] to 8
Thread 2 setting array[5] to 10
Thread 0 setting array[0] to 0
Thread 0 setting array[1] to 2
Thread 1 setting array[2] to 4
Thread 1 setting array[3] to 6
Executed in parallel by multiple threads.
Executed by thread 2
```

n = 3

```
manisha@MIT-DSCA-FL2158:~$ export OMP_NUM_THREADS=3
manisha@MIT-DSCA-FL2158:~$ ./if_clause
Thread 0 setting array[0] to 0
Thread 0 setting array[1] to 2
Thread 0 setting array[2] to 4
Executed serially by a single thread 0
```

OpenMP: num_threads Clause

- The **num_threads** clause in OpenMP is used with the parallel construct to explicitly specify the number of threads that should be used to execute the parallel region.
- This clause provides control over the parallel execution by allowing you to set the number of threads on a per-region basis.
- **#pragma omp parallel num_threads(n)**: This directive specifies that the following block of code should be executed by **n** threads.
- num_threads(scalar-integer-expression)
- The num_threads clause has a higher priority over the omp_set_num_threads() function, which sets the default number of threads for parallel regions.

OpenMP: num_threads Clause

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

int main() {
    int n = 5;
    int array[n];

    // Set the default number of threads
    omp_set_num_threads(4);

    // Parallel region with num_threads clause
    #pragma omp parallel num_threads(2)
    {
        #pragma omp for
        for (int i = 0; i < n; i++) {
            array[i] = i * 2;
            printf("Thread %d setting array[%d] to %d\n", omp_get_thread_num(), i, array[i]);
        }

        #pragma omp single
        {
            printf("This parallel region used %d threads.\n", omp_get_num_threads());
        }
    } // end of parallel region

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~$ ./number_threads
Thread 0 setting array[0] to 0
Thread 0 setting array[1] to 2
Thread 0 setting array[2] to 4
Thread 1 setting array[3] to 6
Thread 1 setting array[4] to 8
This parallel region used 2 threads.
```

OpenMP: num_threads Clause

```c
#include <stdio.h>
#include <omp.h>
int main(){
        int n = 6, TID;
        (void) omp_set_num_threads(4);
#pragma omp parallel if(n>5) num_threads(n) default(none) private(TID) shared(n)
        {
                TID=omp_get_thread_num();
#pragma omp single
                {
                        printf("Value of n = %d\n",n);
                        printf("Number of threads in the parallel region = %d\n",omp_get_num_threads());
                }
                printf("Print statement is executed by thread %d\n",TID);
        }
        return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~$ ./number_threads1
Value of n = 6
Number of threads in the parallel region = 6
Print statement is executed by thread 5
Print statement is executed by thread 3
Print statement is executed by thread 2
Print statement is executed by thread 0
Print statement is executed by thread 1
Print statement is executed by thread 4
```

```
manisha@MIT-DSCA-FL2158:~$ ./number_threads1
Value of n = 5
Number of threads in the parallel region = 1
Print statement is executed by thread 0
```

OpenMP: ordered Clause

- The ordered clause in OpenMP is used to ensure that certain parts of a parallel loop execute in the order of the loop iterations.
- This can be necessary when the order of execution is important for correctness, even in a parallel context.
- It does not take any arguments and is supported on the loop construct only.
- It has to be used if the ordered construct is present in a parallel region, since its purpose is to inform the compiler of the presence of this construct.

OpenMP: ordered Clause

Directive: #pragma omp for ordered

- This directive is used to mark a loop as requiring ordered execution of specific code sections.

Ordered Construct: #pragma omp ordered

- This directive is used within the loop to mark the section of code that must be executed in order.

Performance Penalty: The ordered clause and construct come with a performance penalty

- The OpenMP implementation needs to perform additional book-keeping tasks to keep track of the order in which threads should execute the corresponding region.
- If threads finish out of order, some threads might have to wait, causing further performance penalties.

OpenMP: reduction Clause

- The reduction clause in OpenMP is used to perform certain types of calculations (involving mathematically associative and commutative operators).
  - They can be performed in parallel without code modification.
- Reductions involve combining results from different threads into a single result using associative and commutative operators.
- Common examples include: **sum, product, minimum, and maximum.**

Directive:

- **#pragma omp parallel for reduction(operator:list)**: This directive specifies that the reduction operation will be performed in parallel.
- The operators used in the reduction clause must be mathematically associative and commutative, such as +, *, -, &, |, ^, &&, ||, and their corresponding functions.
- Reductions are common in scientific and engineering programs, where they may be used to test for convergence or to compute statistical data, among other things.

OpenMP: reduction Clause

Reduction Variables:

- **reduction(operator:list)**: The variables involved in the reduction operation are listed after the operator in the reduction clause.
- The results will be shared among threads, and it is unnecessary to explicitly specify them as shared.

Restrictions:

- Aggregate types (arrays), pointer types, and reference types are not supported.
- A reduction variable must not be const-qualified.
- The operator specified in the clause cannot be overloaded with respect to the variables that appear in the clause.

OpenMP: reduction Clause

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

int main() {
    int n = 1000;
    int array[n];
    int sum = 0;

    // Initialize array elements
    for (int i = 0; i < n; i++) {
        array[i] = i + 1; // Array contains numbers from 1 to 1000
    }

    // Parallel region with reduction clause
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += array[i];

    }
    printf("Sum of array elements: %d\n", sum);

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./reduction_clause
Sum of array elements: 500500
```

OpenMP: reduction Clause

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

int main() {
    int n = 1000;
    int array[n];
    int max_value = INT_MIN;

    // Initialize array elements
    for (int i = 0; i < n; i++) {
        array[i] = i + 1; // Array contains numbers from 1 to 1000
    }

    // Parallel region with reduction clause to find the maximum value
    #pragma omp parallel for reduction(max:max_value)
    for (int i = 0; i < n; i++) {
        if (array[i] > max_value) {
            max_value = array[i];
        }
    }

    printf("Maximum value in the array: %d\n", max_value);

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./reduction_clause1
Maximum value in the array: 1000
```

OpenMP: reduction Clause

```c
int a=10;
int i=0;
int sum=0;
#pragma omp parallel private(i) num_threads(6)
{

    #pragma omp for
    for(i=0;i<6;i++)
    {
        sum =sum + a;
        printf("Sum value=%d   ID=%d\n",sum,omp_get_thread_num());
    }

}
printf("Outside Sum value=%d   ID=%d\n",sum,omp_get_thread_num());
```

```
Sum value=10   ID=0
Sum value=20   ID=1
Sum value=20   ID=3
Sum value=30   ID=2
Sum value=40   ID=5
Sum value=50   ID=4
Outside Sum value=50   ID=0

--------------------------------
Process exited after 0.4014 seconds with return value 0
Press any key to continue . . .
```

```c
int a=10;
int i=0;
int sum=0;
#pragma omp parallel private(i) num_threads(6)
{

    #pragma omp for reduction(+:sum)
    for(i=0;i<6;i++)
    {
        sum =sum + a;
        printf("Sum value=%d   ID=%d\n",sum,omp_get_thread_num());
    }

}
printf("Outside Sum value=%d   ID=%d\n",sum,omp_get_thread_num());
```

```
Sum value=10   ID=3
Sum value=10   ID=4
Sum value=10   ID=0
Sum value=10   ID=1
Sum value=10   ID=2
Sum value=10   ID=5
Outside Sum value=60   ID=0

--------------------------------
Process exited after 0.4353 seconds with return value 0
Press any key to continue . . .
```

OpenMP: copyin Clause

- By default, static variables are shared among all threads in a parallel region. This means that if one thread modifies a static variable, other threads will see the modified value.
- Each thread to have its own separate instance of a static variable, rather than sharing a single instance. This can be achieved using the threadprivate directive in OpenMP.
- The threadprivate directive makes a static variable private to each thread. Each thread gets its own copy of the variable, which persists for the lifetime of the thread.
- The copyin clause in OpenMP is used to copy the value of the master thread's threadprivate variables to the corresponding threadprivate variables of other threads in a parallel region.

OpenMP: copyin Clause

- **Undefined Initial Values:** Private variables in OpenMP have undefined initial values. Using copyin, you can initialize these variables with specific values.
- This clause allows for the straightforward initialization of threadprivate variables across all threads in a team.
- The copy is carried out after the team of threads is formed and before the execution of the parallel region starts.
- **Syntax**: **copyin(list)**, where list is a comma-separated list of threadprivate variables.

OpenMP: copyin Clause

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

// Declare threadprivate variables
int x;
#pragma omp threadprivate(x)

int main() {
    x = 42; // Initialize the master thread's x

    // Parallel region with copyin clause
    #pragma omp parallel copyin(x)
    {
        printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
        x = omp_get_thread_num(); // Each thread modifies its own copy of x
    }

    // Master thread prints its x value
    printf("Master thread: x = %d\n", x);

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ export OMP_NUM_THREADS=4
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./copyin_clause
Thread 0: x = 42
Thread 2: x = 42
Thread 1: x = 42
Thread 3: x = 42
Master thread: x = 0
```

OpenMP: copyprivate Clause

- The copyprivate clause is used to broadcast the value of a private variable from one thread to all other threads in the team.
- It is supported on the single directive only.
- It is used in conjunction with the single directive to ensure that one thread initializes the data, which is then copied to other threads.
- After the single construct has ended, the values of variables specified in the copyprivate list are copied to other threads before they leave the associated barrier. This means all threads will synchronize at this point.
- Syntax: copyprivate(list)

  - where list is a comma-separated list of private variables.

OpenMP: copyprivate Clause

```c
int main()
{
    int a=10;

    #pragma omp parallel private(a)
    {
        #pragma omp single
        {
            a=a+5;
        }
        printf("Id--%d    =%d\n",omp_get_thread_num(),a);

    }
    return 0;
}
```

```
Id--1   =5
Id--2   =0
Id--4   =0
Id--3   =0
Id--5   =0
Id--0   =0
Id--7   =0
Id--6   =0

--------------------------------
Process exited after 0.1778 seconds with return value 0
Press any key to continue . . .
```

```c
int main()
{
    int a=10;

    #pragma omp parallel private(a)
    {
        #pragma omp single copyprivate(a)
        {
            a=a+5;
        }
        printf("Id--%d    =%d\n",omp_get_thread_num(),a);

    }
    return 0;
}
```

```
Id--4   =5
Id--3   =5
Id--6   =5
Id--5   =5
Id--0   =5
Id--7   =5
Id--1   =5
Id--2   =5

--------------------------------
Process exited after 0.1717 seconds with return value 0
Press any key to continue . . .
```

Advanced OpenMP Constructs

- These are considered special-purpose because the need to use them strongly depends on the application.
  - **Nested Parallelism**
  - **Flush directive**
  - **Threadprivate directive**
- In OpenMP, nested parallelism refers to the ability of threads in a parallel region to create their own parallel regions.
- For example, certain recursive algorithms can take advantage of nested parallelism in a natural way, but many applications do not need this feature.
- Nested parallelism requires sufficient hardware resources to manage multiple thread teams. On systems with limited resources, excessive nesting might lead to inefficient use of resources.
- The overhead associated with creating and managing multiple levels of parallel regions can outweigh the benefits.

Nested parallelism

- In OpenMP, nested parallelism refers to the ability of threads in a parallel region to create their own parallel regions.
- If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team and becomes the master of that new team
  - This is generally referred to in OpenMP as **“nested parallelism”**
- If nested parallelism is not supported
  - parallel constructs that are nested within other parallel constructs will be ignored.
  - parallel region serialized (executed by a single thread only).
- **Frequent starting and stopping of parallel regions may introduce a non-trivial performance penalty.** It should be used only when the performance benefits outweigh these costs.

Nested parallelism

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

int main() {
    // Disable nested parallelism
    omp_set_nested(0);

    // Set the number of threads for the outer parallel region
    omp_set_num_threads(2);

    #pragma omp parallel
    {
        // Print thread number in the outer parallel region
        int outer_thread_id = omp_get_thread_num();
        printf("Outer thread %d is executing\n", outer_thread_id);

        // Nested parallel region
        #pragma omp parallel
        {
            // Print thread number in the nested parallel region
            int inner_thread_id = omp_get_thread_num();
            printf("  Nested thread %d (from outer thread %d) is executing\n", inner_thread_id, outer_thread_id);
        }
    }

    return 0;
}
```

```
Outer thread 0 is executing
  Nested thread 0 (from outer thread 0) is executing
Outer thread 1 is executing
  Nested thread 0 (from outer thread 1) is executing
```

Nested parallelism

- Nested parallelism must be explicitly enabled in OpenMP using the environment variable **OMP_NESTED** or programmatically using **omp_set_nested().** By default, nested parallelism is disabled to avoid unintended performance penalties.

```
export OMP_NESTED=TRUE  # Enable nested parallelism
export OMP_NESTED=FALSE # Disable nested parallelism
```

- **What will happen if we call omp_get_thread_num() function from the nested region?**
  - It returns the thread id starting from 0 to one less than the number of threads in the current thread team.
  - Thread numbers are no longer unique. (Thread numbers are only unique within the context of the current parallel region's team, not across nested teams).

Nested parallelism

```
omp_set_nested(1);
printf("Nested parallelism is %s\n",omp_get_nested() ? "supported" : "not supported");
#pragma omp parallel
{
    printf("Thread %d executes the outer parallel region\n",
    omp_get_thread_num());
    #pragma omp parallel num_threads(2)
        {
        printf(" Thread %d executes inner parallel region\n",
        omp_get_thread_num());
        } /*-- End of inner parallel region --*/
} /*-- End of outer parallel region --*/
```

```
Nested parallelism is supported
Thread 4 executes the outer parallel region
Thread 7 executes the outer parallel region
Thread 2 executes the outer parallel region
Thread 6 executes the outer parallel region
Thread 3 executes the outer parallel region
Thread 0 executes the outer parallel region
Thread 1 executes the outer parallel region
Thread 5 executes the outer parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
```

Nested parallelism

```
omp_set_nested(1);
printf("Nested parallelism is %s\n",omp_get_nested() ? "supported" : "not supported");
int TID=0;
#pragma omp parallel private(TID)
{
    TID = omp_get_thread_num();
    printf("Thread %d executes the outer parallel region\n",TID);
    #pragma omp parallel num_threads(2) firstprivate(TID)
    {
        printf("TID %d: Thread %d executes inner parallel region
        TID,omp_get_thread_num());
    } /*-- End of inner parallel region --*/
} /*-- End of outer parallel region --*/
```

```
Nested parallelism is supported
Thread 5 executes the outer parallel region
Thread 1 executes the outer parallel region
Thread 6 executes the outer parallel region
Thread 4 executes the outer parallel region
Thread 7 executes the outer parallel region
Thread 3 executes the outer parallel region
Thread 0 executes the outer parallel region
TID 4: Thread 1 executes inner parallel region
TID 5: Thread 0 executes inner parallel region
TID 5: Thread 1 executes inner parallel region
TID 1: Thread 0 executes inner parallel region
TID 1: Thread 1 executes inner parallel region
TID 6: Thread 0 executes inner parallel region
TID 6: Thread 1 executes inner parallel region
TID 4: Thread 0 executes inner parallel region
Thread 2 executes the outer parallel region
TID 7: Thread 0 executes inner parallel region
TID 2: Thread 0 executes inner parallel region
TID 3: Thread 0 executes inner parallel region
TID 3: Thread 1 executes inner parallel region
TID 0: Thread 0 executes inner parallel region
TID 0: Thread 1 executes inner parallel region
TID 7: Thread 1 executes inner parallel region
TID 2: Thread 1 executes inner parallel region
```

Nested parallelism

```c
#include <omp.h>
#include <stdio.h>
#define N 2 // Size of the matrices
void add_matrices(int A[N][N], int B[N][N], int C[N][N]) {
    // Set the number of threads for the outer parallel regio
    omp_set_num_threads(2);
    #pragma omp parallel
    {
        int outer_thread_num = omp_get_thread_num();
        #pragma omp for
        for (int i = 0; i < N; i++) {
            printf("Outer thread %d processing row %d\n", out
            // Set the number of threads for the inner parall
            #pragma omp parallel num_threads(2)
            {
                int inner_thread_num = omp_get_thread_num();
                // Ensure only one thread prints to avoid clu
                printf("Outer thread %d created inner thread
                #pragma omp for
                for (int j = 0; j < N; j++) {
                    C[i][j] = A[i][j] + B[i][j];
                    printf("Inner thread %d computing C[%d][%d] = %d\n", inner_thread_num, i, j, C[i][j]);
                }
            }
        }
    }
}
int main() {
    // Define and initialize matrices A and B
    int A[N][N] = {{1, 2}, {3, 4}};
    int B[N][N] = {{5, 6}, {7, 8}};
    int C[N][N] = {0}; // Result matrix
    // Enable nested parallelism
    omp_set_nested(1);
    // Add matrices
    add_matrices(A, B, C);
    // Print the result matrix
    printf("Result matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

```
Outer thread 0 processing row 0
Outer thread 0 created inner thread 0
Inner thread 0 computing C[0][0] = 6
Outer thread 1 processing row 1
Outer thread 1 created inner thread 0
Inner thread 0 computing C[1][0] = 10
Outer thread 0 created inner thread 1
Inner thread 1 computing C[0][1] = 8
Outer thread 1 created inner thread 1
Inner thread 1 computing C[1][1] = 12
Result matrix C:
6 8
10 12
```

```
Outer thread 0 processing row 0
Outer thread 1 processing row 1
Outer thread 0 created inner thread 0
Inner thread 0 computing C[0][0] = 6
Outer thread 0 created inner thread 1
Inner thread 1 computing C[0][1] = 8
Outer thread 1 created inner thread 0
Inner thread 0 computing C[1][0] = 10
Outer thread 1 created inner thread 1
Inner thread 1 computing C[1][1] = 12
Result matrix C:
6 8
10 12
```

OpenMP: Flush Directive

- OpenMP memory model distinguishes between shared data and private data:
  - Which is accessible and visible to all threads (shared data)
  - Which is local to an individual thread (private data)
- If a thread updates shared data, the new values will first be saved in a register and then stored back to the local cache.
- Other threads doesn't have access to these memories immediately. (i.e., other threads do not immediately see these updates because the changes are not instantly propagated to global shared memory.
- Cache-coherent machines broadcast these shared variables to other threads, making them aware of the changes.

OpenMP: Flush Directive

- OpenMP ensures that all modifications to shared data are eventually written back to the main memory.
- Modifications are thus available to all threads, at synchronization points in the program.
- Between these points, threads can hold new values for shared variables in their local memory rather than global shared memory.
- This approach is known as the Relaxed Consistency Model.

OpenMP: Flush Directive

- Sometimes, threads need to see the updated values of shared variables immediately, before reaching the next synchronization point.
- Flush directive is used to ensure that updates made by a thread to shared variables are visible to other threads.
- The purpose of the flush directive is to make a thread's temporary view of shared data consistent with the values in memory.

```
#pragma omp flush [(list)]
```

  - If a list is provided, only the specified variables are flushed.
  - If no list is provided, all thread-visible shared data is flushed.

OpenMP: Flush Directive

- If a thread has updated variables: The new values are flushed to memory, making them accessible to all other threads.
- If a thread has not updated variables: Ensures that the thread's local copies of the data are replaced by the latest values from main memory.
- This does not synchronize the actions of different threads: rather, it forces the executing thread to make its shared data values consistent with shared memory.
- Since the compiler reorders operations to enhance program performance, one cannot assume that the flush operation will remain exactly in the position, relative to other operations, in which it was placed by the programmer.
- What can be guaranteed is that it will not change its position relative to any operations involving the flushed variables.

OpenMP: Flush Directive

- OpenMP includes implicit flush operations at several key points:
  - At all explicit and implicit barriers (e.g., at the end of a parallel region or worksharing construct).
  - Entry to and exit from critical regions.
  - Entry to and exit from lock routines.

OpenMP: Flush Directive

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

int main() {
    int data, flag = 0;

    #pragma omp parallel num_threads(2)
    {
        if (omp_get_thread_num() == 0) {
            data = 42;
            #pragma omp flush(flag, data)
            /* Set flag to release thread 1 */
            flag = 1;
            #pragma omp flush(flag)
        }
        else if (omp_get_thread_num() == 1) {
            #pragma omp flush(flag, data)
            while (flag < 1) {
                #pragma omp flush(flag, data)
            }
            #pragma omp flush(flag, data)
            printf("flag=%d data=%d\n", flag, data);
        }
    }

    return 0;
}
```

```
manisha@MIT-DSCA-FL2158:~/HPC24/HPC_Class$ ./flush_ex
flag=1 data=42
```

OpenMP: Threadprivate Directive

- By default, global data (static in c) is shared. This means that if one thread modifies a static variable, other threads will see the modified value.
- But in some situations we may need, or would prefer to have, private data that persists throughout the computation.
- This can be achieved using the threadprivate directive in OpenMP.
- Each thread gets a private or “local” copy of the specified global variables.

#pragma omp threadprivate *(list)*

OpenMP: Threadprivate Directive

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

// Global variable
int counter;
#pragma omp threadprivate(counter)

int main() {
    // Initialize counter to 0 in the master thread before any parallel region
    counter = 0;

    // First parallel region
    #pragma omp parallel num_threads(4)
    {
        counter = omp_get_thread_num(); // Initialize counter in each thread
        printf("First parallel region - Thread %d: counter = %d\n", omp_get_thread_num(), counter);
    }

    // Second parallel region
    #pragma omp parallel num_threads(4)
    {
        counter += 10; // Each thread increments its private counter by 10
        printf("Second parallel region - Thread %d: counter = %d\n", omp_get_thread_num(), counter);
    }

    return 0;
}
```

```
First parallel region - Thread 3: counter = 3
First parallel region - Thread 1: counter = 1
First parallel region - Thread 0: counter = 0
First parallel region - Thread 2: counter = 2
Second parallel region - Thread 2: counter = 12
Second parallel region - Thread 0: counter = 10
Second parallel region - Thread 3: counter = 13
Second parallel region - Thread 1: counter = 11
```

OpenMP: Threadprivate Directive

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

int main() {
    int counter = 0;

    // First parallel region
    #pragma omp parallel firstprivate(counter) num_threads(4)
    {
        counter = omp_get_thread_num(); // Initialize counter in each thread
        printf("First parallel region - Thread %d: counter = %d\n", omp_get_thread_num(), counter);
    }

    // Second parallel region
    #pragma omp parallel firstprivate(counter) num_threads(4)
    {
        counter += 10; // Each thread increments its private counter by 10
        printf("Second parallel region - Thread %d: counter = %d\n", omp_get_thread_num(), counter);
    }

    return 0;
}
```

```
First parallel region - Thread 0: counter = 0
First parallel region - Thread 1: counter = 1
First parallel region - Thread 3: counter = 3
First parallel region - Thread 2: counter = 2
Second parallel region - Thread 1: counter = 10
Second parallel region - Thread 3: counter = 10
Second parallel region - Thread 2: counter = 10
Second parallel region - Thread 0: counter = 10
```