

Course name: High Performance Computing

Course code: DSE 3122

No of contact hours/week: 3

Credit: 3

By

Sandhya Parasnath Dubey and Manisha

Assistant Professor,

Department of Data Science and Computer Applications - MIT, Manipal

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

Mob: +91-9886542135

Email: sandhyadubey24@manipal.edu

**"Our greatest weakness lies in giving up. The most certain way to succeed is
always to try just one more time"**

OpenMP: Introduction

- OpenMP (Open Multi-Processing) is an API (Application Programming Interface)
 - Supports **multi-platform shared memory multiprocessing** programming in C, C++, and Fortran.
- It allows developers to write parallel programs that can run efficiently on **multicore systems, shared-memory systems, and distributed-memory systems**.

Key features of OpenMP include:

- 1. Directive-based Programming Model:** OpenMP uses **compiler directives** to specify parallel regions in the code where multiple threads will execute concurrently.
- 2. Thread-level Parallelism:** It provides mechanisms to **create and manage threads**, distributing the workload among them to achieve parallel execution.
- 3. Portability:** OpenMP is supported by many compilers across different platforms, making it widely accessible and portable.

OpenMP: Introduction

- 4. **Ease of Use:** Compared to other parallel programming models, OpenMP is relatively easy to learn and integrate into existing codebases because it uses pragma directives to indicate parallelism.
- 5. **Scalability:** OpenMP programs can scale well with increasing numbers of cores or processors, provided the workload can be efficiently divided among threads.
- 6. **Support for Nested Parallelism:** OpenMP supports nested parallel regions, allowing finer-grained control over parallel execution.

Developers typically use OpenMP for tasks that can be parallelized across multiple threads, such as loops and sections of code that can be executed independently.

By utilizing OpenMP directives, programmers can take advantage of the computing power of modern multicore processors without needing to manage low-level threading details explicitly.

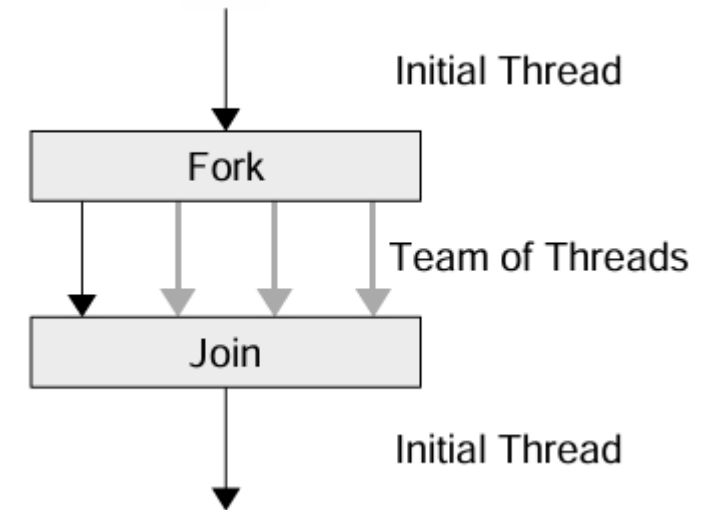
Overall, OpenMP is a powerful tool for parallel programming in shared-memory environments, offering a balance between simplicity, performance, and portability.

OpenMP: Introduction

- A **thread** is a runtime entity that is able to independently execute a stream of instructions.
- OpenMP builds on a large body of work that supports the specification of programs for execution by a collection of cooperating threads
- The individual threads need just a few resources of their own: a **program counter and an area in memory** to save variables that are specific to it (including registers and a stack).

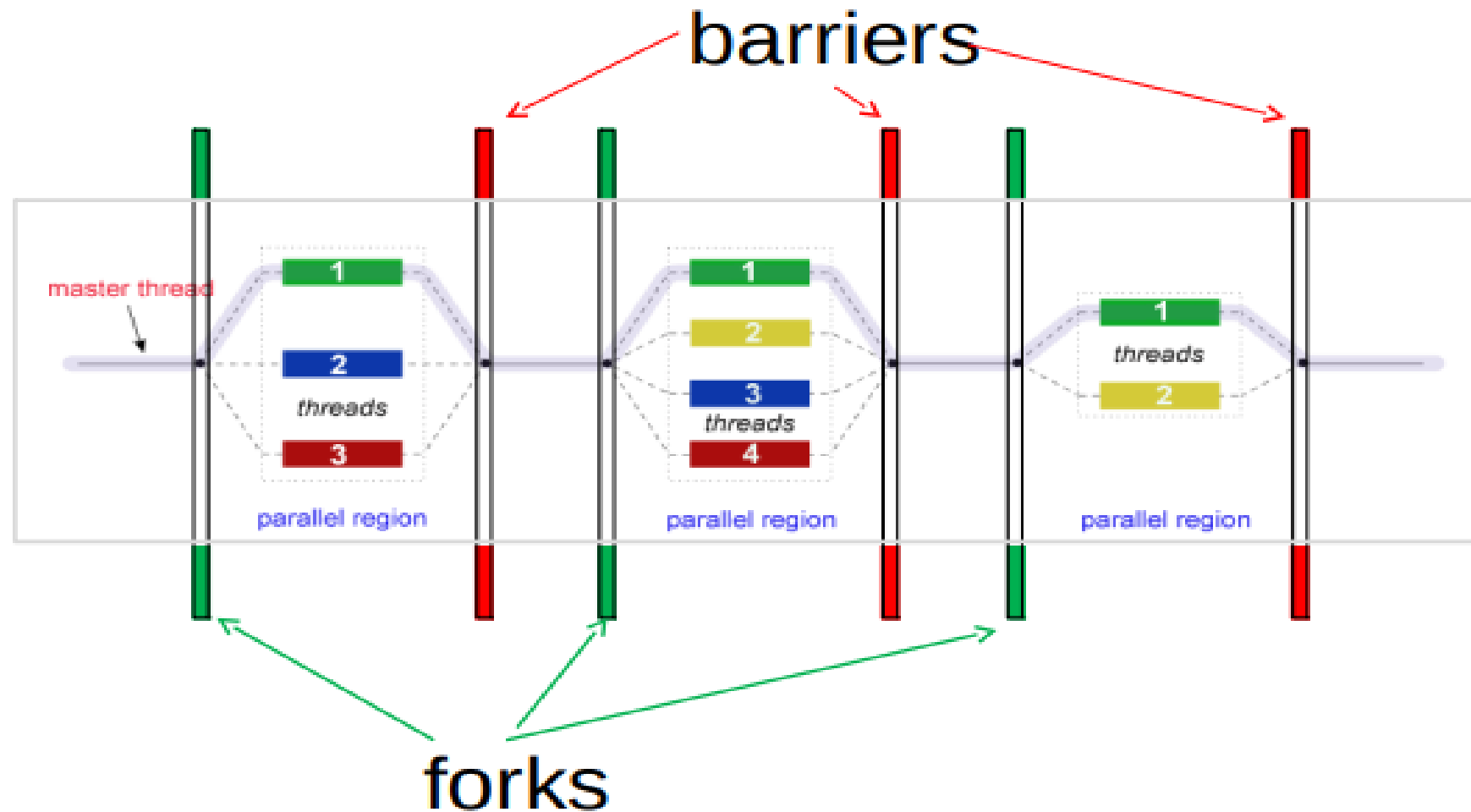
The fork-join programming model

- ✓ The program starts as a single thread of execution, the **initial thread**. A team of threads is forked at the beginning of a parallel region and joined at the end
- ✓ Whenever an OpenMP parallel construct is encountered by a thread while it is executing the program, **it creates a team of threads (this is the fork)**, becomes the master of the team, and collaborates with the other members of the team to execute the code dynamically enclosed by the construct.
- ✓ At the end of the construct, only the original thread, or master of the team, continues; all others terminate (this is the join). Each portion of code enclosed by a **parallel construct** is called a parallel region.



OpenMP: Introduction

Fork-join model



OpenMP: Introduction

What OpenMP Isn't

- OpenMP doesn't check for **data dependencies, data conflicts, deadlocks, or race conditions**. You are responsible for avoiding those yourself
- OpenMP doesn't check for **non-conforming** code sequences
- OpenMP doesn't **guarantee identical behavior** across vendors or hardware, or even between multiple runs on the same vendor's hardware
- OpenMP doesn't guarantee the **order in which threads execute**, just that they do execute
- OpenMP is not **overhead-free**
- OpenMP does not prevent you from writing code that triggers **cache performance problems**

OpenMP: Introduction

OpenMP Components

Directives

- Parallel region
- Worksharing constructs
- Tasking
- Offloading
- Affinity
- Error Handling
- SIMD
- Synchronization
- Data-sharing attributes

Runtime Environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Schedule
- Active levels
- Thread limit
- Nesting level
- Ancestor thread
- Team size
- Locking
- Wallclock timer

Environment Variable

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
- Stacksize
- Idle threads
- Active levels
- Thread limit

OpenMP: Introduction

OpenMP Components

- ✓ An **OpenMP directive** is a specially **formatted comment** or **pragma** that generally applies to the executable code immediately following it in the program.
- ✓ A directive or OpenMP routine generally affects only those threads that encounter it.
- ✓ Many of the directives are applied to a structured block of code, a sequence of executable statements with a single entry at the top and a single exit at the bottom in Fortran programs, and an executable statement in C/C++ (which may be a compound statement with a single entry and single exit).

OpenMP: Introduction

OpenMP Components

- ✓ OpenMP provides means for the user to
 - ✓ create teams of threads for parallel execution,
 - ✓ specify how to share work among the members of a team,
 - ✓ declare both shared and private variables, and
 - ✓ synchronize threads and enable them to perform certain operations exclusively (i.e., without interference by other threads).

OpenMP is commonly used to **incrementally parallelize an existing sequential code**, and this task is most easily accomplished by creating parallel regions one at a time.

OpenMP: Introduction

OpenMP Components

- ✓ Creating Teams of Threads
 - ✓ A team of threads is created to execute the code in a parallel region of an OpenMP program.
 - ✓ To accomplish this, the programmer simply specifies the parallel region by inserting a parallel directive immediately before the code that is to be executed in parallel to mark its start
- ✓ Additional information can be supplied along with the parallel directive.
- ✓ This is mostly used to enable threads to have private copies of some data for the duration of the parallel region and to initialize that data.
- ✓ At the end of a parallel region is an implicit barrier synchronization: this means that no thread can progress until all other threads in the team have reached that point

OpenMP: Introduction

Sharing Work among Threads

- ✓ The OpenMP work-sharing **directives** state **how the computation in a structured block of code is to be distributed** among the threads.
 - ✓ Unless explicitly overridden by the programmer, an **implicit barrier synchronization** also exists at the end of a work-sharing construct.

Work Sharing and Loops

- ✓ **for (C/C++) loop** among the threads in a team.
- ✓ The programmer inserts the **appropriate directive immediately before each loop within a parallel region** that is to be shared among threads.
- ✓ Work-sharing directives **cannot be applied to all kinds of loops** that occur in C/C++ code.

OpenMP: Introduction

Sharing Work among Threads

Work Sharing and Loops

- ✓ All OpenMP strategies for sharing the work in loops **assign one or more disjoint sets of iterations to each thread.**
- ✓ The programmer may specify the method used to partition the iteration set.
- ✓ The most straightforward strategy assigns **one contiguous chunk of iterations to each thread.**
- ✓ More complicated strategies include dynamically computing the next chunk of iterations for a thread.
- ✓ If the programmer does not provide a strategy, then an implementation-defined default will be used.

OpenMP: Introduction

The OpenMP Memory Model

- ✓ Based on the **shared-memory model**; hence, by default, **data is shared among the threads and is visible to all of them.**
- ✓ Data can be **declared to be shared or private** with respect to a parallel region or work-sharing construct.
- ✓ Threads need a place to store their private data at run time. For this, each thread has its own special region in memory known as the **thread stack**.

OpenMP: Introduction

Thread Synchronization

- ✓ Synchronizing, or coordinating the actions of, threads is necessary in order to ensure the proper ordering of their accesses to shared data and to prevent data corruption.

Performance Considerations

- ✓ If we denote by T_1 the execution time of an application on 1 processor,
 - ✓ then in an ideal situation, the execution time on P processors should be T_1/P .
 - ✓ If T_P denotes the execution time on P processors, then the ratio $S = T_1/T_P$ is referred as the **parallel speedup** and is a measure for the success of the parallelization.

OpenMP: Introduction

Sharing Work among Threads

- ✓ Virtually all programs contain **some regions that are suitable** for parallelization and other regions that are not.
- ✓ By using an increasing number of processors, the time spent in the parallelized parts of the program is reduced, but the sequential section remains the same.
- ✓ Eventually the execution time is completely dominated by the time taken to compute the sequential portion, which **puts an upper limit on the expected speedup. This effect, known as Amdahl's law**

$S = 1/(f_{\text{par}}/P + (1-f_{\text{par}}))$ where f_{par} is the parallel fraction of the code and P is the number of processors. In the ideal case when all of the code runs in parallel, $f_{\text{par}} = 1$, the **expected speedup is equal to the number of processors.**

- ✓ If only 80% of the code runs in parallel ($f_{\text{par}} = 0.8$), the maximal speedup one can expect on 16 processors is 4 and on 32 processors is 4.4.

OpenMP: Writing a First OpenMP Program

Objective:

- ✓ familiarize with the OpenMP syntax and give a few basic rules.
- ✓ Explain how OpenMP can be used to parallelize an existing application in a manner that preserves the original sequential version.

OpenMP: Writing a First OpenMP Program

- The impact of OpenMP parallelization is frequently localized
- Original sequential version is preserved
- For C and C++ programs, **pragmas** are provided by the OpenMP API to control parallelism.
- In OpenMP these are called **directives**. They always start with **#pragma omp**, followed by a specific keyword that identifies the directive, with possibly one or more so-called clauses, each separated by a comma.
- OpenMP directives in C/C++ are **case-sensitive**.

```
#pragma omp directive-name [clause[[,] clause]...] new-line
```

OpenMP: Writing a First OpenMP Program

“Hello Word” Example/1

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

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

    printf("Hello World\n");

    return (0) ;
}
```

OpenMP: Writing a First OpenMP Program

“Hello Word” 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);
}
```

OpenMP: Writing a First OpenMP Program

“Hello Word” Example/2

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

int main()
{
    #pragma omp parallel
    {
        printf("Hello World\n");
    } //End of parallel region
return(0);
}
```

```
"PHello.c" 12L, 168C written
sandhya@telnet:~/PP$ gcc PHello.c
sandhya@telnet:~/PP$ ./a.out
Hello World
sandhya@telnet:~/PP$
```

```
sandhya@telnet:~/PP$ gcc -fopenmp PHello.c
sandhya@telnet:~/PP$ ./a.out
Hello World
Hello World
Hello World
Hello World
sandhya@telnet:~/PP$
```

OpenMP: Writing a First OpenMP Program

“Hello Word” Example/2

```
$ 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);
}
```

[illegible]

OpenMP: Introduction

“Hello Word” Example

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

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

    return(0);
}
```

Directives

Runtime Environment

OpenMP: Introduction

“Hello Word” Example

```
$  
$ 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

OpenMP: Introduction

OpenMP controlling number of threads

- Once a program is compiled, the number of threads can be controlled using the following shell variables

- **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)
```

OpenMP: Introduction

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: Introduction

Hello World in OpenMP

- Each thread has a unique integer “id”; master thread has “id” 0
- Other threads have “id” 1, 2, ...
- OpenMP runtime function

`omp_get_thread_num()`

returns a thread’s unique “id”.

- What will be the programs output?

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

OpenMP: Introduction

A sample OpenMP program

```
main( )
{
    omp_set_num_threads( 8 );
    #pragma omp parallel default(none)
    {
        printf( "Hello, World, from thread #%-d ! \n" , omp_get_thread_num( ) );
    }
    return 0;
}
```

Ch.4 OpenMPLanguage Features

- Objective
 - how OpenMP constructs and clauses are used to tackle some programming problems
 - Parallel Construct
 - Work-Sharing Constructs
 1. Loop Construct
 2. Sections Construct
 3. Single Construct
 4. Workshare Construct (Fortran only)
 - Data-Sharing, No Wait, and Schedule Clauses

Ch.4 OpenMPLanguage Features

- **OpenMP Directive:** In C/C++, a #pragma and in Fortran, a comment, that specifies OpenMP **program behavior**.
- **Executable directive:** An OpenMP directive that is **not declarative**; that is, it may be placed in an executable context.
- **Construct:** An OpenMP executable directive (and, for Fortran, the paired end directive, if any) and the associated statement, loop, or structured block, if any, not including the code in any called routines, that is, the lexical extent of an executable directive

Ch.4 OpenMPLanguage Features

- Parallel Construct

```
#pragma omp parallel [clause[[, clause]. . .]  
    structured block
```

- Syntax of the parallel construct in C/C++. The parallel region **implicitly ends at the end of the structured block**. This is a closing curly brace (}) in most cases.
- In OpenMP: a program without a parallel construct will be executed sequentially

Ch.4 OpenMPLanguage Features

- Parallel Construct

- ❑ A team of **threads is created** to execute the associated parallel region, which is the code dynamically contained within the parallel construct.
- ❑ But although this construct ensures that computations are performed in parallel, **it does not distribute the work** of the region among the threads in a team.
- ❑ In fact, if the programmer does not use the appropriate syntax to specify this action, the **work will be replicated**.
- ❑ At the end of a parallel region, there is an **implied barrier** that forces all threads to wait until the work inside the region has been completed.
- ❑ **Only the initial thread continues execution after the end of the parallel region**

Ch.4 OpenMPLanguage Features

- Parallel Construct

- ☐ The thread that encounters the parallel construct becomes the **master of the new team**.
- ☐ Each thread in the team is assigned a **unique thread number** (also referred to as the “thread id”) to identify it.
- ☐ They **range from zero (for the master thread) up to one less than the number of threads within the team**, and they can be accessed by the programmer.
- ☐ OpenMP library function **omp_get_thread_num()** is used to obtain the number of each thread executing the parallel region

Ch.4 OpenMPLanguage Features

- Parallel Construct

```
#pragma omp parallel
{
```

```
    printf("The parallel region is executed by thread %d\n",
           omp_get_thread_num());
```

```
    if ( omp_get_thread_num() == 2 ) {
        printf("  Thread %d does things differently\n",
               omp_get_thread_num());
    }
```

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

The parallel region is executed by thread 0

The parallel region is executed by thread 3

The parallel region is executed by thread 2

Thread 2 does things differently

The parallel region is executed by thread 1

Example of a parallel region: All threads execute the first printf statement, but only the thread with thread number 2 executes the second one.

Ch.4 OpenMPLanguage Features

diff between constructs and clause we need to know

#pragma omp parallel and all are directives(sections also) the names along with it like if num_threads shared private and all are clauses

- Parallel Construct: **Clauses supported by the parallel construct**

| | |
|---|-----------|
| if (<i>scalar-expression</i>) | (C/C++) |
| if (<i>scalar-logical-expression</i>) | (Fortran) |
| num_threads (<i>integer-expression</i>) | (C/C++) |
| num_threads (<i>scalar-integer-expression</i>) | (Fortran) |
| private (<i>list</i>) | |
| firstprivate (<i>list</i>) | |
| shared (<i>list</i>) | |
| default (none shared) | (C/C++) |
| default (none shared private) | (Fortran) |
| copyin (<i>list</i>) | |
| reduction (<i>operator:list</i>) | (C/C++) |
| reduction (<i>{ operator intrinsic_procedure_name } :list</i>) | (Fortran) |

Ch.4 OpenMPLanguage Features

- Parallel Construct
 - At most **one if clause** can appear on the directive.
 - At most **one num threads** clause can appear on the directive. The expression for the clause must evaluate to a positive integer value.
 - A parallel region is **active** if it is **executed by a team of threads** consisting of more than one thread.
 - If it is executed by one thread only, it has been serialized and is considered to be **inactive**.
 - For example, one can specify that a parallel region be conditionally executed, in order to be sure that it contains enough work for this to be worthwhile

OpenMP: Introduction

Sharing Work among Threads

Divide for-loop for parallel sections

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



```
#pragma omp parallel
{
    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;
}
```

// Assume number of threads=4

Thread 0

```
Id=0;
x[0]=0;
x[4]=0;
```

Thread 1

```
Id=1;
x[1]=0;
x[5]=0;
```

Thread 2

```
Id=2;
x[2]=0;
x[6]=0;
```

Thread 3

```
Id=3;
x[3]=0;
x[7]=0;
```

Ch.4 OpenMPLanguage Features

- **Sharing the Work among Threads** in an OpenMP Program

| Functionality | Syntax in C/C++ | Syntax in Fortran |
|---|-----------------------------|-------------------------|
| Distribute iterations over the threads | #pragma omp for | !\$omp do |
| Distribute independent work units | #pragma omp sections | !\$omp sections |
| Only one thread executes the code block | #pragma omp single | !\$omp single |
| Parallelize array-syntax | | !\$omp workshare |

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program
 - Specifies a region of code whose work is to be distributed among the executing threads;
 - it also specifies the manner in which the work in the region is to be parceled out.
 - A work-sharing region must bind to an active parallel region in order to have an effect.
 - If a work sharing directive is encountered in an inactive parallel region or in the sequential part of the program, it is simply ignored.

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program
 - The two main rules regarding work-sharing constructs are as follows:
 - Each work-sharing region 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 a team.

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program

Loop Construct : The loop construct causes the iterations of the loop immediately following it to be executed in parallel.

At run time, the loop iterations are distributed across the threads.

```
#pragma omp for [clause[,] clause]...  
for-loop
```

Use of this construct is limited to those kinds of loops where the number of iterations can be counted; that is, the loop must have an integer counter variable whose value is incremented (or decremented) by a fixed amount at each iteration until some specified upper (or lower) bound is reached.

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program

`for (init-expr ; var relop b ; incr-expr)`

```
#pragma omp parallel shared(n) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
        printf("Thread %d executes loop iteration %d\n",
               omp_get_thread_num(),i);
} /*-- End of parallel region --*/
```

Figure 4.10: **Example of a work-sharing loop** – Each thread executes a subset of the total iteration space $i = 0, \dots, n - 1$.

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

Figure 4.11: **Output from the example shown in Figure 4.10** – The example is executed for $n = 9$ and uses four threads.

OpenMP: Introduction

Sharing Work among Threads

Divide for-loop for parallel sections

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



```
#pragma omp parallel
{
    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;
}
```

// Assume number of threads=4

Thread 0

```
Id=0;
x[0]=0;
x[4]=0;
```

Thread 1

```
Id=1;
x[1]=0;
x[5]=0;
```

Thread 2

```
Id=2;
x[2]=0;
x[6]=0;
```

Thread 3

```
Id=3;
x[3]=0;
x[7]=0;
```

OpenMP: Introduction

Sharing Work among Threads

Use `pragma parallel for`

```
for (int i=0; i<8; i++) x[i]=0;
```



```
#pragma omp parallel for  
{  
    for (int i=0; i<8; i++)  
        x[i]=0;  
}
```

System divides loop iterations to threads

```
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;
```

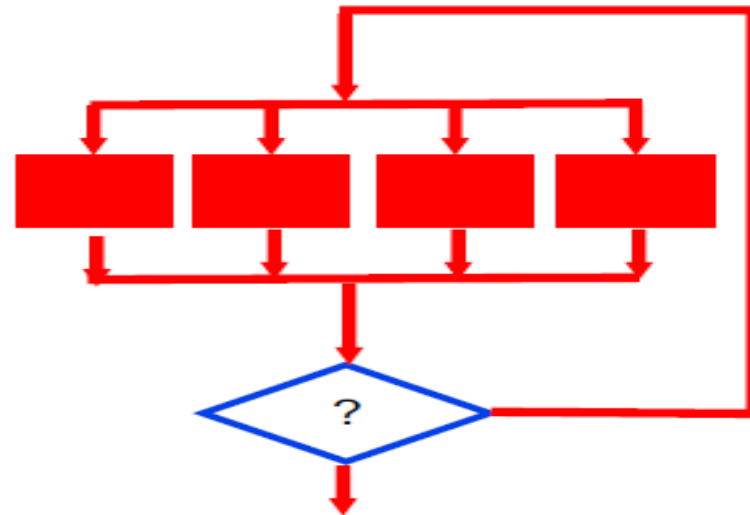
OpenMP: Introduction

Sharing Work among Threads

Programming Model – Parallel Loops

- Requirement for parallel loops
 - No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor calculates loop bounds and divide iterations among parallel threads

```
#pragma omp parallel for
for( i=0; i < 25; i++ )
{
    printf("Foo");
}
```




OpenMP: Introduction

Sharing Work among Threads

Example

```
for (i=0; i<max; i++) zero[i] = 0;
```

- Breaks *for loop* into chunks, and allocate each to a separate thread
 - e.g. if `max = 100` with 2 threads:
assign 0-49 to thread 0, and 50-99 to thread 1
 - Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
 - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
 - No premature exits from the loop allowed 
 - i.e. No `break`, `return`, `exit`, `goto` statements
- In general, don't jump outside of any pragma block**

OpenMP: Introduction

Sharing Work among Threads

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: Introduction

Sharing Work among Threads

```
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: Introduction

Sharing Work among Threads

```
private(list)  
firstprivate(list)  
lastprivate(list)  
reduction(operator:list) (C/C++)  
reduction({ operator | intrinsic_procedure_name } : list) (Fortran)  
ordered  
schedule (kind[, chunk_size])  
nowait
```

Figure 4.13: **Clauses supported by the loop construct** – They are described in Section 4.5 and Section 4.8.

OpenMP: Introduction

Sharing Work among Threads

```
#pragma omp parallel shared(n,a,b) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
        a[i] = i;

    #pragma omp for
    for (i=0; i<n; i++)
        b[i] = 2 * a[i];
} /*-- End of parallel region --*/
```

Two work-sharing loops in one parallel region

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program

Sections construct: is the easiest way to get different threads to carry out different kinds of work.

Since it permits us to specify several different code regions, **each of which will be executed by one of the threads.**

It consists of two directives: **#pragma omp sections** in C/C++ to indicate the **start of the construct** and second, the **#pragma omp section** directive in C/C++ to mark **each distinct section.**

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program

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

Syntax of the sections construct in C/C++— The number of sections controls, and limits, the amount of parallelism. If there are “n” of these code blocks, at most “n” threads can execute in parallel.

can execute max n but we can define more but won't be used

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        (void) funcA();

        #pragma omp section
        (void) funcB();
    } /*-- End of sections block --*/
} /*-- End of parallel region --*/
```

Figure 4.16: **Example of parallel sections** – If two or more threads are available, one thread invokes `funcA()` and another thread calls `funcB()`. Any other threads are idle.

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program

```
void funcA()
{
    printf("In funcA: this section is executed by thread %d\n",
        omp_get_thread_num());
}
```

Output

```
In funcA: this section is executed by thread 0
In funcB: this section is executed by thread 1
```

This construct might lead to a load-balancing problem

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program

```
private(list)  
firstprivate(list)  
lastprivate(list)  
reduction(operator:list) (C/C++)  
reduction({ operator | intrinsic_procedure_name } :list) (Fortran)  
nowait
```

Clauses supported by the sections construct

Ch.4 OpenMPLanguage Features

- Sharing the Work among Threads in an OpenMP Program

The Single Construct

- Specifies that this block should be executed by one thread only.
- It does not state which thread should execute the code block;
- indeed, the thread chosen could vary from one run to another. It can also differ for different single constructs within one application.
- This construct should really be used when we do not care which thread executes this part of the application, as long as the work gets done by exactly one thread.
- The other threads wait at a barrier until the thread executing the single code block has completed.

Ch.4 OpenMP Language Features

- Sharing the Work among Threads in an OpenMP Program

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

Figure 4.20: Syntax of the single construct in C/C++ – Only one thread executes the structured block.

There is an implicit barrier at the end of the single construct.

Ch.4 OpenMP Language Features

| |
|--|
| <code>private(<i>list</i>)</code> |
| <code>firstprivate(<i>list</i>)</code> |
| <code>copyprivate(<i>list</i>)</code> |
| <code>nowait</code> |

```
#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<n; i++)
        b[i] = a;
} /*-- End of parallel region --*/

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

Single construct executed by thread 3
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

Figure 4.22: **Example of the single construct** – Only one thread initializes the shared variable a.

Ch.4 OpenMP Language Features

| Full version | Combined construct |
|--|--|
| <pre>#pragma omp parallel { #pragma omp for for-loop }</pre> | <pre>#pragma omp parallel for for-loop</pre> |
| <pre>#pragma omp parallel { #pragma omp sections { [#pragma omp section] structured block [#pragma omp section structured block] ... } }</pre> | <pre>#pragma omp parallel sections { [#pragma omp section] structured block [#pragma omp section structured block] ... }</pre> |

Syntax of the combined constructs in C/C++

Readability

Performance advantage: compiler knows what to expect and may be able to generate slightly more efficient code. For example, it will not insert more than one barrier at the end of the region.

Ch.4 OpenMPLanguage Features:Clauses

- **Shared:** Specify which data will be shared among the threads executing the region it is associated with.
- One unique instance of these variables, and each thread can freely read or modify the values.
- The syntax for this clause is **shared(list)**. All items in the list are data objects that will be shared among the threads in the team.

```
#pragma omp parallel for shared(a)
for (i=0; i<n; i++)
{
    a[i] += i;
} /*-- End of parallel for --*/
```

Alert: multiple threads might attempt to simultaneously update the same memory location or that one thread might try to read from a location that another thread is updating

OpenMP: data sharing

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

int main()
{
    int a=5;
    #pragma omp parallel shared(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
25
15
outside:  25
sandhya@telnet:~/PP$ ./a.out
15
35
15
25
outside:  35
sandhya@telnet:~/PP$ ./a.out
25
15
15
35
outside:  35
```

Ch.4 OpenMPLanguage Features

- Private

Changes made to the data by one thread are not visible to other threads

syntax is `private(list)`

```
#pragma omp parallel for private(i,a)
for (i=0; i<n; i++)
{
    a = i+1;
    printf("Thread %d has a value of a = %d for i = %d\n",
        omp_get_thread_num(),a,i);
} /*-- End of parallel for --*/
```

```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
```

The results are for n = 5, using three threads to execute the code

Example of the private clause– Each thread has a local copy of variables i and a

OpenMP: Data sharing

Example: **private** clause

```
#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
```


OpenMP: data sharing:Private

- Private
 - The values of private data are undefined upon entry to and exit from the specific construct.
 - The value of any variable with the same name as the private variable in the enclosing region is also undefined after the construct has terminated, even if the corresponding variable was defined prior to the region.
 - Since this point may be unintuitive, care must be taken to check that the code respects this.

OpenMP: data sharing: Lastprivate Clause

- What if the value of `a` is needed after the loop?
- We have just stated that the values of data specified in the private clause can no longer be accessed after the corresponding region terminates.
- OpenMP offers a workaround if such a value is needed.
- The `lastprivate clause` addresses this situation; it is supported on the work-sharing loop and sections constructs.

OpenMP: data sharing: Lastprivate Clause

- syntax is `lastprivate(list)`
- It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution.
- In a parallel program, however, we must explain what “last” means.
- In the case of its use with a work-shared loop, the object will have the value from the iteration of the loop that would be last in a sequential execution.
- If the lastprivate clause is used on a sections construct, the object gets assigned the value that it has at the end of the lexically last sections construct.

OpenMP: data sharing: Lastprivate Clause

```
#pragma omp parallel for private(i) lastprivate(a)
for (i=0; i<n; i++)
{
    a = i+1;
    printf("Thread %d has a value of a = %d for i = %d\n",
        omp_get_thread_num(),a,i);
} /*-- End of parallel for --*/

printf("Value of a after parallel for: a = %d\n",a);
```

Figure 4.34: **Example of the lastprivate clause** – This clause makes the sequentially last value of variable **a** accessible outside the parallel loop.

```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
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
```

Figure 4.35: **Output from the example shown in Figure 4.34** – Variable *n* is set to 5, and three threads are used. The last value of variable **a** corresponds to the value for *i* = 4, as expected.

OpenMP: data sharing: Lastprivate Clause

```
#pragma omp parallel for private(i) private(a) shared(a_shared)
  for (i=0; i<n; i++)
  {
      a = i+1;
      printf("Thread %d has a value of a = %d for i = %d\n",
            omp_get_thread_num(),a,i);
      if ( i == n-1 ) a_shared = a;
  } /*-- End of parallel for --*/
```

Figure 4.36: **Alternative code for the example in Figure 4.34** – This code shows another way to get the behavior of the `lastprivate` clause. However, we recommend use of the clause, not something like this.

- A performance penalty is likely to be associated with the use of `lastprivate`, because the OpenMP library needs to keep track of which thread executes the last iteration.

OpenMP: data sharing: Firstprivate Clause

- Private data is also **undefined on entry to the construct** where it is specified.
- This could be a problem if we need to pre-initialize private variables with values that are available prior to the region in which they will be used.
- OpenMP provides the **firstprivate** construct to help out in such cases.
- Variables that are declared to be “firstprivate” are **private variables, but they are pre-initialized with the value of the variable with the same name before the construct.**
- The initialization is carried out by the initial thread prior to the execution of the construct.
- The firstprivate clause is supported on the parallel construct, plus the work-sharing loop, sections, and single constructs.

OpenMP: Data sharing

Example: **firstprivate** clause

```
#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$
```

OpenMP: data sharing: Firstprivate Clause

- The syntax is **firstprivate(list)**

```
for(i=0; i<vlen; i++) a[i] = -i-1;

indx = 4;
#pragma omp parallel default(none) firstprivate(indx) \
        private(i,TID) shared(n,a)
{
    TID = omp_get_thread_num();

    indx += n*TID;
    for(i=indx; i<indx+n; i++)
        a[i] = TID + 1;
} /*-- End of parallel region --*/

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

After the parallel region:

```
a[0] = -1
a[1] = -2
a[2] = -3
a[3] = -4
a[4] = 1
a[5] = 1
a[6] = 2
a[7] = 2
a[8] = 3
a[9] = 3
```

Figure 4.38: Output from the program shown in Figure 4.37: The initial offset into the vector is set to `indx = 4`. Variable `n = 2` and three threads are used. Therefore the total length of the vector is given by $vlen = 4 * 2 * 3 = 10$. The first `indx = 4` values of vector `a` are not initialized

Figure 4.37: **Example using the firstprivate clause** – Each thread has a pre-initialized copy of variable `indx`. This variable is still private, so threads can update it individually.

OpenMP: data sharing: Default Clause

- Used to give variables a **default data-sharing attribute**
- For example, **default(shared)** assigns the **shared attribute to all variables referenced in the construct.**
- The syntax in C/C++ is given by **default (none | shared).**
- **#pragma omp for default(shared) private(a,b,c),** for example, declares all variables to be shared, with the exception of a, b, and c.
- **If default(none) is specified instead, the programmer is forced to specify a data-sharing attribute for each variable in the construct.**

OpenMP: data sharing: Nowait Clause

- Allows the programmer to fine-tune a program's performance
- When we introduced the work-sharing constructs, we mentioned that there is an implicit barrier at the end of them. This clause overrides that feature of OpenMP
- If it is added to a construct, the barrier at the end of the associated construct will be suppressed.
- The barrier at the end of a parallel region cannot be suppressed

OpenMP: data sharing: Nowait Clause

- When a thread is finished with the work associated with the parallelized for loop, it continues and no longer waits for the other threads to finish as well.

```
#pragma omp for nowait
for (i=0; i<n; i++)
{
    .....
}
```

OpenMP: Nowait clause

```
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\Nowait_trial.exe
To automatically close the console when debugging stops, enable Tool Output when debugging stops.
Press any key to close this window . . .

```
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\Nowait_trial.exe
To automatically close the console when debugging stops, enable Tool Output when debugging stops.
Press any key to close this window . . .

OpenMP: Data sharing

Clauses to Control Parallel and Work-Sharing Constructs

- Shared
- Private
- Lastprivate
- Firstprivate
- Default
- Nowait
- Schedule

OpenMP: Schedule Clause

- The schedule clause is supported on the **loop construct only**
- It is used to *control the manner in which 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])**
- The granularity of this workload distribution is a chunk
 - A contiguous, nonempty subset of the iteration space
- **Note** that the chunk_size parameter need not be a constant

OpenMP: Schedule Clause

- **Static:**

- Iterations are divided into chunks of size chunk size
- The chunks are assigned to the threads statically in a **round-robin** manner, in the **order of the thread number**
- 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
- **The most straight forward schedule is static. It also has the least overhead and is the default on many OpenMP compilers, to be used in the absence of an explicit schedule clause.**

- **Dynamic:**

- The iterations are assigned to threads as the threads request them
- 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

OpenMP: Schedule Clause

- **Guided**

- The iterations are assigned to threads as the threads request them
- 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.
- For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations, divided by the number of threads, decreasing to 1.
- For a chunk size of “k” ($k > 1$), chunks do not contain fewer than k iterations

OpenMP: Schedule Clause

- 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.**
- The rationale behind this scheme is that initially larger chunks are desirable because they reduce the overhead
- It is not always easy to select the appropriate schedule and value for *chunk size* up front.
- The choice may depend (among other things) not only on the code in the loop but also on the specific problem size and the number of threads used

OpenMP: Schedule Clause

- **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

OpenMP: Schedule Clause

```
#pragma omp parallel for default(none) schedule(runtime) \
                        private(i,j) shared(n)
for (i=0; i<n; i++)
{
    printf("Iteration %d executed by thread %d\n",
        i, omp_get_thread_num());
    for (j=0; j<i; j++)
        system("sleep 1");
} /*-- End of parallel for --*/
```

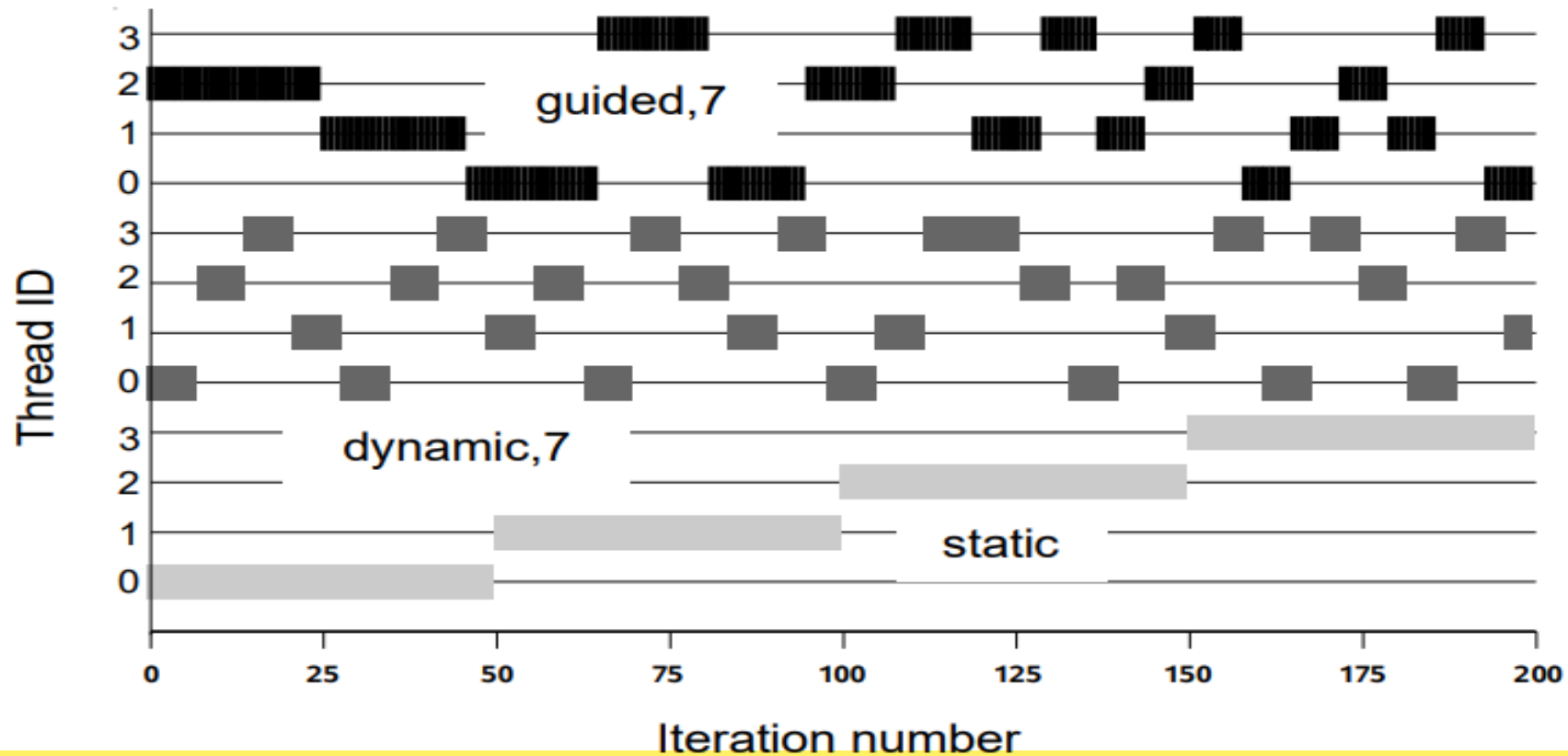
Figure 4.43: Example of the schedule clause – The runtime variant of this clause is used. The `OMP_SCHEDULE` environment variable is used to specify the schedule that should be used when executing this loop.

OpenMP: Schedule Clause

Table 4.1: Example of various workload distribution policies – The behavior for the `dynamic` and `guided` scheduling policies is nondeterministic. A subsequent run with the same program may give different results.

| Iteration | TID static | TID static,2 | TID dynamic | TID dynamic,2 | TID guided | TID guided,2 |
|-----------|---------------|-----------------|----------------|------------------|---------------|-----------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 1 | 3 | 3 | 3 | 3 |
| 3 | 1 | 1 | 2 | 3 | 2 | 3 |
| 4 | 1 | 2 | 1 | 2 | 1 | 2 |
| 5 | 2 | 2 | 0 | 2 | 0 | 2 |
| 6 | 2 | 3 | 3 | 1 | 3 | 1 |
| 7 | 3 | 3 | 2 | 1 | 2 | 1 |
| 8 | 3 | 0 | 1 | 0 | 1 | 0 |

OpenMP: Schedule Clause



The static schedule is most efficient from a performance point of view, since dynamic and guided have higher overheads.

OpenMP: Schedule Clause

```
#pragma omp parallel for schedule(static,20) num_threads(5)
for (int i = 0; i < 100; i++)
{
    printf("Loop executed by thread id=%d\n", omp_get_thread_num());
}
```

```
#pragma omp parallel for schedule(dynamic,20) num_threads(5)
for (int i = 0; i < 100; i++)
{
    printf("Loop executed by thread id=%d\n", omp_get_thread_num());
}
```

```
#pragma omp parallel for schedule(runtime) num_threads(5)
for (int i = 0; i < 100; i++)
{
    printf("Loop executed by thread id=%d\n", omp_get_thread_num());
}
```

OpenMP: Data sharing

Clauses to Control Parallel and Work-Sharing Constructs

- Shared
- Private
- Lastprivate
- Firstprivate
- Default
- Nowait
- Schedule

OpenMP: Synchronization Constructs

- An algorithm may require us to manipulate the actions of multiple threads to ensure that updates to a shared variable occur in a certain order
- It may simply need to ensure that two threads do not simultaneously attempt to write a shared object
- Synchronization Constructs can be used when implicit barrier are not sufficient enough

Helps in organised accesses of shared data by multiple threads

OpenMP: Synchronization Constructs

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

Helps in organised accesses of shared data by multiple threads

Synchronization Constructs: Barrier

- A barrier is a point in the execution of a program where threads wait for each other
 - no thread in the team of threads it applies to may proceed beyond a barrier until all threads in the team have reached that point
- The compiler automatically inserts a barrier at the end of the construct
- 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.

```
#pragma omp barrier
```

Synchronization Constructs: Barrier

```
#pragma omp parallel private(TID)
{
    TID = omp_get_thread_num();
    if (TID < omp_get_num_threads()/2 ) system("sleep 3");
    (void) print_time(TID,"before");

    #pragma omp barrier

    (void) print_time(TID,"after ");
} /*-- End of parallel region --*/
```

```
Thread 2 before barrier at 01:12:05
Thread 3 before barrier at 01:12:05
Thread 1 before barrier at 01:12:08
Thread 0 before barrier at 01:12:08
Thread 1 after  barrier at 01:12:08
Thread 3 after  barrier at 01:12:08
Thread 2 after  barrier at 01:12:08
Thread 0 after  barrier at 01:12:08
```

Figure 4.48: **Output from the example in Figure 4.47** – Four threads reach the barrier at different times. Note that threads 2 and 3 wait for three seconds in the barrier.

Figure 4.47: **Example usage of the barrier construct** – A thread waits at the barrier until the last thread in the team arrives. To demonstrate this behavior, we have made sure that some threads take longer than others to reach this point.

Synchronization Constructs: Barrier

```
int main()
{
    int TID;
    # pragma omp parallel private(TID)
    {
        TID=omp_get_thread_num();
        printf("Number of Parallel threads running are: %d\n",omp_get_num_threads());
        if (TID<omp_get_num_threads()/2)
        {
            system("sleep 5");

            printf("My TID is %d and I am going to sleep for 5s Sleeping \n",TID);
        }
        printf("My TID is %d Reached barrier.....waiting\n",TID);
    # pragma omp barrier
    printf("After barrier.....\n");
    }

    printf("outside parallel region \n");
    printf("Number of Parallel threads running are: %d\n",omp_get_num_threads());
    return 0;
}
```

Synchronization Constructs: Barrier

```
root@MAHE-L480-40087:~/sandhya/PP# ./a.out
My TID is 2 Reached barrier.....waiting
My TID is 3 Reached barrier.....waiting
My TID is 0 and I am going to sleep for 5s Sleeping
My TID is 0 Reached barrier.....waiting
My TID is 1 and I am going to sleep for 5s Sleeping
My TID is 1 Reached barrier.....waiting
After barrier..... My TID is:0.
After barrier..... My TID is:1.
After barrier..... My TID is:3.
After barrier..... My TID is:2.
outside parallel region
Number of Parallel threads running are: 1
root@MAHE-L480-40087:~/sandhya/PP#
```

Synchronization Constructs: Barrier

```
#pragma omp parallel
{
    int TID;
    TID = omp_get_thread_num();
    if (TID < omp_get_num_threads() / 2)
    {
        for (int i = 0; i < 9000; i++)
        {
        }
        printf("Sleeping\n");
    }
    printf("Reached barrier....waiting\n");
    #pragma omp barrier
    printf("After barrier....\n");
}
return 0;
```

```
Sleeping
Sleeping
Reached barrier....waiting
Reached barrier....waiting
Reached barrier....waiting
Reached barrier....waiting
Reached barrier....waiting
Reached barrier....waiting
Sleeping
Reached barrier....waiting
Reached barrier....waiting
Sleeping
Reached barrier....waiting
After barrier....
After barrier....
After barrier....
After barrier....
After barrier....
After barrier....
After barrier....
After barrier....

D:\DSCA\Parallel_Computing\Lab_Programs\Barrier_exe
.
To automatically close the console when debugging s
le when debugging stops.
Press any key to close this window . . .
```

Synchronization Constructs: Barrier

- The most common use for a barrier is to avoid a **data race condition**
- Inserting a barrier between the **writes to** and **reads from a shared variable** guarantees that the accesses are appropriately ordered

Synchronization Constructs: Ordered

- Allows one to execute a structured block within a parallel loop in sequential order
- This is sometimes used, for instance, to enforce an ordering on the printing of data computed by different threads
- Help determine whether there are any data races in the associated code

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

Ordered clause is there and ordered construct is also there so keep that in mind

Synchronization Constructs: Ordered

```
#pragma omp parallel default(none) shared(n,a,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",sum);
return 0;
}
```

Synchronization Constructs: Ordered

```
root@MAHE-L480-40087:~/sandhya/PP# ./a.out
Enter the value of n
11
TID =0: sumlLocal=0: sum=0
TID =0: sumlLocal=1: sum=1
TID =0: sumlLocal=3: sum=4
TID =1: sumlLocal=3: sum=7
TID =1: sumlLocal=7: sum=14
TID =1: sumlLocal=12: sum=26
TID =2: sumlLocal=6: sum=32
TID =2: sumlLocal=13: sum=45
TID =2: sumlLocal=21: sum=66
TID =3: sumlLocal=9: sum=75
TID =3: sumlLocal=19: sum=94
outside parallel region
value of sum after parallel region:94
root@MAHE-L480-40087:~/sandhya/PP#
```

Synchronization Constructs: Ordered

```
//Ordered Clause
int n = 8;
int a[8] = {};
#pragma omp parallel for default(none) ordered schedule(runtime) shared(n,a)
for (int 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]);
    }
}
```

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

```
D:\DSCA\Parallel_Computing\Lab_Programs\Barr
.
To automatically close the console when debug
le when debugging stops.
Press any key to close this window . . .
```

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

```
#pragma omp critical [(name)]  
    structured block
```

Synchronization Constructs: Critical

```
#pragma omp parallel default(none) shared(n,a,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 region \n");
printf("value of sum after parallel region:%d\n",sum);

return 0;
}
```

Synchronization Constructs: Critical

```
root@MAHE-L480-40087:~/sandhya/PP# ./a.out
Enter the value of n
11
TID =0: sumlLocal=0: sum=0
TID =0: sumlLocal=1: sum=1
TID =0: sumlLocal=3: sum=4
TID =1: sumlLocal=3: sum=7
TID =1: sumlLocal=7: sum=14
TID =1: sumlLocal=12: sum=26
TID =3: sumlLocal=9: sum=35
TID =3: sumlLocal=19: sum=54
TID =2: sumlLocal=6: sum=60
TID =2: sumlLocal=13: sum=73
TID =2: sumlLocal=21: sum=94
outside parallel region
value of sum after parallel region:94
root@MAHE-L480-40087:~/sandhya/PP#
```

```
root@MAHE-L480-40087:~/sandhya/PP# ./a.out
Enter the value of n
11
TID =0: sumlLocal=0: sum=0
TID =0: sumlLocal=1: sum=1
TID =0: sumlLocal=3: sum=4
TID =1: sumlLocal=3: sum=7
TID =1: sumlLocal=7: sum=14
TID =1: sumlLocal=12: sum=26
TID =2: sumlLocal=6: sum=32
TID =2: sumlLocal=13: sum=45
TID =2: sumlLocal=21: sum=66
TID =3: sumlLocal=9: sum=75
TID =3: sumlLocal=19: sum=94
outside parallel region
value of sum after parallel region:94
root@MAHE-L480-40087:~/sandhya/PP#
```

Critical Vs. Ordered

```
int main()
{
    int n,i,TID,sum=0;
    printf("Enter the value of n\n");
    scanf("%d",&n);
    int a[n];
#pragma omp parallel default(none) shared(n,a,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 region \n");
    printf("value of sum after parallel region:%d\n",sum);
}
```

Synchronization Constructs: Critical

```
root@MAHE-L480-40087:~/sandhya/PP# ./a.out
Enter the value of n
8
TID =0: sumLocal=1: sum=1
TID =1: sumLocal=5: sum=6
TID =2: sumLocal=9: sum=15
TID =3: sumLocal=13: sum=28
outside parallel region
value of sum after parallel region:28
root@MAHE-L480-40087:~/sandhya/PP#
```


Synchronization Constructs: Critical

```
int sum = 1;
int n = 8;
int a[8] = {};
#pragma omp parallel shared(n,a,sum)
{
    int TID = omp_get_thread_num();
    int sumLocal = 1;
    #pragma omp for
    for (int i = 0; i < n; i++)
        sumLocal += a[i];
    #pragma omp critical
    {
        sum += sumLocal;
        printf("TID=%d: sumLocal=%d sum = %d\n", TID, sumLocal, sum);
    }
}
printf("Value of sum after parallel region: %d\n", sum);
```

```
TID=5: sumLocal=1 sum = 2
TID=7: sumLocal=1 sum = 3
TID=6: sumLocal=1 sum = 4
TID=3: sumLocal=1 sum = 5
TID=2: sumLocal=1 sum = 6
TID=4: sumLocal=1 sum = 7
TID=1: sumLocal=1 sum = 8
TID=0: sumLocal=1 sum = 9
Value of sum after parallel region: 9
```

```
D:\DSCA\Parallel_Computing\Lab_Programs\Barrier_example
To automatically close the console when debugging stops
le when debugging stops.
Press any key to close this window . . .
```

Synchronization Constructs: Atomic

- The atomic construct, which also enables multiple threads to update shared data without interference, can be an efficient alternative to the critical region
- It is applied only to the (single) assignment statement that immediately follows it
- The reason it is applied to just one assignment statement is that it protects updates to an individual memory location

```
#pragma omp atomic  
statement
```

Synchronization Constructs: Atomic

- If a thread is atomically updating a value, then no other thread may do so simultaneously
- This restriction applies to all threads that execute a program, not just the threads in the same team

```
int atomic_read(const int* x)
{
    int value;
    #pragma omp atomic read
    value = *x;
    return value;
}
void atomic_write(int* x, int value)
{
    #pragma omp atomic write
    *x = value;
}
```

Synchronization Constructs: Atomic

```
#pragma omp parallel default(none) shared(n,a,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 atomic
        // {
        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",sum);
return 0;
```

Synchronization Constructs: Atomic

```
root@MAHE-L480-40087:~/sandhya/PP# ./a.out
Enter the value of n
11
TID =1: sumlLocal=3: sum=9
TID =1: sumlLocal=7: sum=25
TID =1: sumlLocal=12: sum=37
TID =3: sumlLocal=9: sum=18
TID =3: sumlLocal=19: sum=56
TID =2: sumlLocal=6: sum=6
TID =2: sumlLocal=13: sum=69
TID =2: sumlLocal=21: sum=90
TID =0: sumlLocal=0: sum=9
TID =0: sumlLocal=1: sum=91
TID =0: sumlLocal=3: sum=94
outside parallel region
value of sum after parallel region:94
root@MAHE-L480-40087:~/sandhya/PP#
```

Synchronization Constructs: Locks

- A thread lock is an object that can be held by at most one thread at a time
- An OpenMP lock can be in one of the following states:
 - **Uninitialized; Unlocked; or Locked**
- If a lock is in the unlocked state, a task can set the lock, which changes its state to locked
- The thread that sets the lock is then said to own the lock
- A thread that owns a lock can unset that lock, returning it to the unlocked state

Synchronization Constructs: Locks

- The general procedure to use locks is as follows:
 - Define the lock variables using **omp_lock_t**
 - Initialize the lock via a call to **omp_init_lock()**
 - Set the lock using **omp_set_lock()** or **omp_test_lock()**. The latter checks whether the lock is actually available before attempting to set it. It is useful to achieve asynchronous thread execution
 - Unset a lock after the work is done via a call to **omp_unset_lock()**
 - Remove the lock association via a call to **omp_destroy_lock()**

Synchronization Constructs: Locks

```
omp_lock_t A;

omp_init_lock(&A);
int b=1;
int c=0;
int d=0;
#pragma omp parallel for
for (int i = 0; i < 10; i++)
{
    // some stuff
    d=d+b;
    printf("##### thread...%d...d=%d\n", omp_get_thread_num(),d);
    omp_set_lock(&A);
    c=c+b;
    printf("Executed by thread...%d...c=%d\n", omp_get_thread_num(),c);
    omp_unset_lock(&A);
    // some stuff
}

omp_destroy_lock(&A);
```

```
##### thread...2...d=1
##### thread...4...d=2
##### thread...3...d=3
##### thread...7...d=5
##### thread...0...d=5
##### thread...6...d=7
##### thread...1...d=8
##### thread...5...d=6
Executed by thread...2...c=1
Executed by thread...4...c=2
Executed by thread...3...c=3
Executed by thread...7...c=4
Executed by thread...0...c=5
##### thread...0...d=9
Executed by thread...6...c=6
Executed by thread...1...c=7
##### thread...1...d=10
Executed by thread...5...c=8
Executed by thread...0...c=9
Executed by thread...1...c=10

-----
Process exited after 0.4338 seconds with return value 0
Press any key to continue . . .
```


Synchronization Constructs: Master

- The master construct defines a block of code that is guaranteed to be executed by the master thread only
- It is thus similar to the single construct
- The master construct is technically not a work-sharing construct, however, and it does not have an implied barrier on entry or exit
- If the master construct is used to initialize data, for example, care needs to be taken that this initialization is completed before the other threads in the team use the data

Synchronization Constructs: Master

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

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

Master construct is 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 0.1579 seconds with return value 0

Press any key to continue . . .

OpenMP: Synchronization Constructs

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

Helps in organised accesses of shared data by multiple threads

OpenMP: If Clause

- The **if clause** is supported on **the parallel construct** only, where it is used to specify conditional execution
- 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

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: Num_threads Clause

- The num threads clause is supported **on the parallel construct only**
- Can be used to specify how many threads should be in the team executing the parallel region
- `num_threads(scalar-integer-expression)`
- Has **higher priority** over `omp_set_num_threads(4)`

OpenMP: Num_threads Clause

```
(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 parallel region: %d\n",
            omp_get_num_threads());
    }
    printf("Print statement executed by thread %d\n",TID);
} /*-- End of parallel region --*/
```

Figure 4.72: Example of the num_threads clause – This clause is used in a parallel region to control the number of threads used.

```
Value of n = 5
Number of threads in parallel region: 1
Print statement executed by thread 0
Value of n = 10
Number of threads in parallel region: 10
Print statement executed by thread 0
Print statement executed by thread 4
Print statement executed by thread 3
Print statement executed by thread 5
Print statement executed by thread 6
Print statement executed by thread 7
Print statement executed by thread 8
Print statement executed by thread 9
Print statement executed by thread 2
Print statement executed by thread 1
```

Figure 4.73: Output of the program given in Figure 4.72

OpenMP: Ordered Clause

- It does not take any arguments and is supported on the loop construct only
- It has to be given if the **ordered construct** is used in a parallel region, since its purpose is to inform the compiler of the presence of this construct

```
//Ordered Clause
int n = 8;
int a[8] = {};
#pragma omp parallel for default(none) ordered schedule(runtime) shared(n,a)
for (int 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]);
    }
}
```

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

D:\DSCA\Parallel_Computing\Lab_Programs\Barr
-
To automatically close the console when debu
le when debugging stops.
Press any key to close this window . . .
```


OpenMP: Ordered Clause

- 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. Moreover, if threads finish out of order, there may be an additional performance penalty because some threads might have to wait.

OpenMP: Reduction Clause

- OpenMP provides the reduction clause for specifying some forms of recurrence calculations (involving mathematically associative and commutative operators)
 - They can be performed in parallel without code modification
- The programmer must identify the operations and the variables that will hold the result values

reduction(operator :list)

- The results will be shared and it is not necessary to specify the corresponding variables explicitly as “shared.”
- The order in which thread-specific values are combined is unspecified

OpenMP: Reduction Clause

- 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.

How Reduction Works:

- **Private Copies:** Each thread gets a private copy of the reduction variable (sum in this case).
- **Partial Computation:** Each thread performs its part of the computation and stores the result in its private copy.
- **Combination:** At the end of the parallel region, all private copies are combined using the specified reduction operator.

OpenMP: Reduction Clause

```
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 . . .
```

```
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: Reduction Clause

| Operator | Initialization value |
|----------|----------------------|
| + | 0 |
| * | 1 |
| - | 0 |
| & | ~0 |
| | 0 |
| ^ | 0 |
| && | 1 |
| | 0 |

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

OpenMP: Reduction Clause

Important Considerations

- **Initial Value:** OpenMP initializes the private copies of the reduction variable according to the operator. For +, the initial value is 0; for *, it is 1, etc.
- **Data Sharing:** The reduction variable is shared across all threads in the final combination phase but private during the computation phase.

OpenMP: Copyin Clause

- Allows us to copy the value of the master thread's **threadprivate variable(s)** to the corresponding threadprivate variables of the other threads
- Threadprivate:
 - Static variables are generally shared by default
 - We can change this by using **threadprivate** clause (We will discuss this later)
- The initial values of private variables are undefined
- The copy is carried out after the team of threads is formed and prior to the start of execution of the parallel region, so that it enables a straightforward initialization of this kind of data object.
- The clause is supported on the parallel directive and the combined parallel work-sharing directives. The syntax is `copyin(list)`.

OpenMP: Copyprivate Clause

- The **copyprivate clause** is supported on the **single directive** only
- It provides a mechanism for broadcasting the value of a private variable from one thread to the other threads in the team
- **Uses:** one thread read or initialize private data that is subsequently used by the other threads as well
- After the single construct has ended, but before the threads have left the associated barrier, the values of variables specified in the associated list are copied to the other threads

OpenMP: Copyprivate Clause

```
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;
}
```

```
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;
}
```

OpenMP: Copyprivate Clause

```
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 . . .
```

```
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.
- 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**
 - **Flush directive**
 - **Threadprivate directive**

Nested parallelism

- 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**

Nested parallelism

- Nested parallelism is not enabled by default in OpenMP. To enable it, you need to set the `OMP_NESTED` environment variable to `TRUE` or call the `omp_set_nested` function in your code.
- `omp_set_nested(1); // Enable nested parallelism`
- Performance: Nested parallelism can introduce significant overhead due to the creation and management of additional threads. It is crucial to ensure that the additional parallelism justifies the overhead.
- Thread Limits: The total number of threads created can grow quickly, leading to resource exhaustion. For example, if you have 2 levels of parallelism with 4 threads each, you can end up with 16 threads in total. Control the number of threads at each level to avoid excessive resource use.

```
export OMP_NESTED=TRUE
```

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

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 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\n",
            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
```


OpenMP: collapse Clause

- Used with **loop constructs** to indicate that **multiple nested loops should be collapsed into a single loop for parallel execution.**
- This can improve performance by increasing the available parallelism and balancing the workload more evenly among threads.

```
#pragma omp parallel for collapse(n)
for (int i = 0; i < N1; ++i) {
    for (int j = 0; j < N2; ++j) {
        // Loop body
    }
}
```

- Here, n is the number of nested loops to collapse.

OpenMP: collapse Clause: How It Works

- When you specify the collapse(n) clause, OpenMP treats the nested loops as a single loop with a larger iteration space.
- This allows the OpenMP runtime to distribute the iterations more evenly across the available threads, potentially improving load balancing and reducing the overhead of managing nested parallel regions.

OpenMP: collapse Clause

- In this example, each iteration of the outer loop is executed by a separate thread, and then the inner loop runs serially within each thread.

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

int main() {
    int N1 = 4, N2 = 4;
    #pragma omp parallel for
    for (int i = 0; i < N1; ++i) {
        for (int j = 0; j < N2; ++j) {
            printf("Thread %d: i = %d, j = %d\n", omp_get_thread_num(), i, j);
        }
    }
    return 0;
}
```

```
Thread 2: i = 2, j = 0
Thread 2: i = 2, j = 1
Thread 2: i = 2, j = 2
Thread 2: i = 2, j = 3
Thread 3: i = 3, j = 0
Thread 3: i = 3, j = 1
Thread 3: i = 3, j = 2
Thread 3: i = 3, j = 3
Thread 0: i = 0, j = 0
Thread 0: i = 0, j = 1
Thread 0: i = 0, j = 2
Thread 0: i = 0, j = 3
Thread 1: i = 1, j = 0
Thread 1: i = 1, j = 1
Thread 1: i = 1, j = 2
Thread 1: i = 1, j = 3
```

OpenMP: collapse Clause

- With the collapse clause, the nested loops are collapsed into a single loop.

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

int main() {
    int N1 = 4, N2 = 4;
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < N1; ++i) {
        for (int j = 0; j < N2; ++j) {
            printf("Thread %d: i = %d, j = %d\n", omp_get_thread_num(), i, j);
        }
    }
    return 0;
}
```

sandhya@DESKTOP-3ERBGCG

```
Thread 0: i = 0, j = 0
Thread 0: i = 0, j = 1
Thread 2: i = 1, j = 0
Thread 2: i = 1, j = 1
Thread 7: i = 3, j = 2
Thread 7: i = 3, j = 3
Thread 6: i = 3, j = 0
Thread 6: i = 3, j = 1
Thread 3: i = 1, j = 2
Thread 3: i = 1, j = 3
Thread 1: i = 0, j = 2
Thread 1: i = 0, j = 3
Thread 5: i = 2, j = 2
Thread 5: i = 2, j = 3
Thread 4: i = 2, j = 0
Thread 4: i = 2, j = 1
```

OpenMP: collapse Clause

- **Benefits of Using collapse**
 - **Increased Parallelism:** Collapsing loops can increase the number of iterations available for parallel execution, improving parallel efficiency, especially when the outer loop has a small number of iterations.
 - **Better Load Balancing:** By treating nested loops as a single loop, OpenMP can distribute the iterations more evenly among threads, leading to better load balancing and potentially reducing idle times.
 - **Reduced Overhead:** Reducing the number of nested parallel regions can decrease the overhead associated with managing these regions, leading to better performance.

OpenMP: collapse Clause

- Example with Uneven Iteration Space

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

int main() {
    int N1 = 4, N2 = 2;
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < N1; ++i) {
        for (int j = 0; j < N2; ++j) {
            printf("Thread %d: i = %d, j = %d\n", omp_get_thread_num(), i, j);
        }
    }
    return 0;
}
```

```
Thread 0: i = 0, j = 0
Thread 3: i = 1, j = 1
Thread 2: i = 1, j = 0
Thread 6: i = 3, j = 0
Thread 7: i = 3, j = 1
Thread 4: i = 2, j = 0
Thread 1: i = 0, j = 1
Thread 5: i = 2, j = 1
```

- the total iteration count is $N1 * N2 = 4 * 2 = 8$, which is evenly distributed among the threads.

OpenMP: collapse Clause

- The collapse clause in OpenMP is a powerful tool for **enhancing parallel performance**, especially when dealing with **nested loops**.
- By collapsing loops, you can increase the available parallelism, achieve better **load balancing**, and **reduce the overhead** associated with managing nested parallel regions.
- This can lead to significant performance improvements in parallel applications.

OpenMP: Flush Directive

- Used to enforce **memory consistency** between threads by making sure that the values of shared variables are updated and visible to all threads.
- It acts as a synchronization point for memory, ensuring that all threads have a consistent view of shared data.
- Purpose of flush
 - Memory Consistency: Ensures that the latest values of shared variables are visible to all threads.
 - Data Visibility: Guarantees that changes made by one thread to shared variables are seen by other threads.
 - Synchronization: Can be used to create a synchronization point without necessarily blocking the execution of threads.

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


OpenMP: Flush Directive

- Without variable-list: When no variables are specified, the flush directive applies to all shared variables.
- With variable-list: When specific variables are listed, the flush directive applies only to those variables.

OpenMP: Flush Directive

- Important Points

- Implicit Flush: Certain OpenMP constructs, such as barriers, critical sections, and atomic operations, imply a flush. Thus, explicit flush directives are often not needed in these contexts.
- Flush and Synchronization: While flush ensures memory consistency, it does not synchronize the execution of threads like barriers do. Threads can continue executing independently after a flush.
- Use Cases: flush is typically used in more complex synchronization scenarios, such as producer-consumer patterns, where explicit memory consistency is crucial without the overhead of full synchronization.
- By using the flush directive appropriately, you can ensure that shared variables are consistently updated and visible across all threads, facilitating correct and efficient parallel programming in OpenMP.

OpenMP: Flush Directive

```
#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;
}
```

OpenMP: Threadprivate Directive

- Used to specify that global variables, file scope variables, and static variables inside a function should be replicated, so each thread gets its own copy of the variable. This is useful when you need each thread to maintain its own state independently of other threads.
- The threadprivate directive allows threads to have their own separate instances of variables, which can be useful in scenarios where you want to maintain thread-specific data.

OpenMP: Threadprivate Directive

- Scope: The threadprivate directive can be applied to global variables, file scope variables, and static variables within functions.
- Automatic variables (local variables with automatic storage duration) cannot be threadprivate.
- Initialization: A threadprivate variable is uninitialized by default for each thread unless explicitly initialized in a serial section before the parallel region.
- Each thread's instance of the variable retains its value between parallel regions within the same thread.
- Consistency Across Regions: If a thread re-enters a parallel region, it retains the value of its threadprivate variables from the previous parallel region.
- Copying Initial Values: To copy the value of a threadprivate variable from the master thread to all other threads at the start of a parallel region, you can use the copyin clause.

OpenMP: Threadprivate Directive

- By default, global data is shared
- Each thread gets a **private** or “**local**” copy of the specified global variables

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

- By default, the threadprivate copies are not allocated or defined

OpenMP: Threadprivate Directive

```
#include <stdlib.h>
#include <omp.h>
/*int tid;
#pragma omp threadprivate(tid)*/
int main()
{
    int numt,tid;
#pragma omp parallel
    {
        int j;
        tid=omp_get_thread_num();
        if(tid==0)
        {
            for(j=0;j<100000000;j++);
            numt=omp_get_num_threads();
        }
    }
#pragma omp parallel
    {
        printf("Hello from thread %d of %d.\n",tid,numt);
    }
    return 0;
}
```

OpenMP: Threadprivate Directive

[illegible]

OpenMP: Threadprivate Directive

```
#include <stdlib.h>
#include <omp.h>
/*int tid;
#pragma omp threadprivate(tid)*/
int main()
{
    int numt,tid;
#pragma omp parallel private(tid)
    {
        int j;
        tid=omp_get_thread_num();
        if(tid==0)
        {
            for(j=0;j<10000000;j++);
            numt=omp_get_num_threads();
        }
    }
#pragma omp parallel
    {
        printf("Hello from thread %d of %d.\n",tid,numt);
    }
    return 0;
}
```

[illegible]

OpenMP:How you measure wall clock time

- Use `omp_get_wtime()`
- Used to measure elapsed wall-clock time.
- This is useful for timing the execution of specific sections of code, particularly when you're interested in evaluating the performance of parallel regions.
- Wall-Clock Time: `omp_get_wtime()` returns the elapsed wall-clock time in seconds.
- This is the real-world time, not CPU time.
- High-Resolution Timer: It provides a high-resolution timer, which is suitable for performance measurements.
- Start and End Times: You typically call `omp_get_wtime()` at the start and end of the section of code you want to measure, then subtract the start time from the end time to get the elapsed time.

OpenMP:How you measure wall clock time

- Performance Measurement: It's ideal for measuring the performance of specific code sections, especially in parallel programs where you want to compare the performance of parallel vs. serial execution.
- Platform Independence: Unlike other timing functions that may depend on the operating system, `omp_get_wtime()` is part of the OpenMP standard, making it portable across different platforms.
- Granularity: The precision of `omp_get_wtime()` can vary depending on the system's timer resolution, but it is generally suitable for most timing purposes in parallel computing.
- Thread-Independent: `omp_get_wtime()` is not affected by the number of threads or the specific thread that calls it, making it reliable for timing parallel sections.
- `omp_get_wtime()` is a simple yet powerful tool in OpenMP for measuring elapsed time. It allows you to time the execution of parallel regions or any section of code, making it an essential function for performance analysis and optimization in parallel programming.

