4.Combined Parallel Work-Sharing Constructs

- OpenMP provides 3 directives :
  - Parallel for
  - Parallel sections
- These directives behave identically to an individual parallel directive being immediately followed by a separate work-sharing directive.

4.Combined Parallel Work-Sharing Constructs

```
#include <omp.h>
#define N 1000
#define CHINKSIZE 100

main(int argc, char *argv[])
{
        int i, chunk;
        float a[N], b[N], c[N];

        /* Some initializations */
        for (i=0; i<N; i++)
              a[i] = b[i] = i*1.0;
        chunk = CHUNKSIZE;
```

```
#pragma omp parallel for \
        shared(a,b,c,chunk) private(i) \
        schedule(static,chunk)
        for(i=0; i<N; i++)
              c[i] = a[i] + b[i];

}
```

5.Task Construct

Purpose:

- It defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team.
- The data environment of the task is determined by the data sharing atribute clauses
- Task execution is subjected to task scheduling

5. Task Construct

#pragma omp task [clause ...] newline

if (scalar expression)
final (scalar expression)
untied
default (shared | none)
mergeable
private (list)
firstprivate (list)
shared (list)

structured_block

6.Synchronization Constructs

Purpose:

- Consider a simple example where 2 threads on 2 different processors are trying to increment a variable x at the same time (assume x is initially 0):

6.Synchronization Constructs

THREAD 1:	THREAD 2:
increment (x)	increment (x)
{	{
x = x + 1;	x = x + 1;
}	}
THREAD 1:	THREAD 2:
10 LOAD A, (x address)	10 LOAD A, (x address)
20 ADD A, 1	20 ADD A, 1
30 STORE A, (x, address)	30 STORE A, (x, address)

6.Synchronization Constructs

- One possible execution sequence:

1. Thread 1 loads the value of x into register A
2. Thread 2 loads the value of x into register A
3. Thread 1 adds 1 to register A
4. Thread 2 adds 1 to register A
5. Thread 1 stores register A at location x
6. Thread 2 stores register A at location x

The resultant value of x will be 1, not 2 as it should be.

6.Synchronization Constructs

- To avoid a situation like this, the incrementing of x must be synchronized between the 2 threads to ensure that the correct result is produced.
- OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads.

6.1 Synchronization Constructs : Master Directive

Purpose:

- It specifies a region that is to be executed only by the master thread of the team.
- All the other threads on the team skip this section of code.
- There is no implied barrier associated with this directive.

```
#pragma omp master newline

Structured_block
```

6.2 Synchronization Constructs : critical Directive

Purpose:

- It specifies a region that is to be executed by only one thread at a time
- If a thread is currently executing inside a critical region and another thread reaches that critical region and attempts to execute it, it will block until the first thread exits that critical region.
- The **optional name** enables multiple different critical regions to exist:
  - Names act as global identifiers. Different critical regions with the same name are treated as the same region
  - All critical sections which are unnamed, are treated as the same section

6.2 Synchronization Constructs : critical Directive

- **Restrictions**:

It is illegal to branch into or out of a critical block.

```
#pragma omp critical [name] newline

Structured_block
```

6.2 Synchronization Constructs : critical Directive

```
#include <omp.h>

main(int argc, char *argv[])
{
        int x;
        x = 0;

    #pragma omp parallel  shared(x)
    {
      #pragma omp critical
            x = x + 1;
    } /* end of parallel region */

}
```

6.3 Synchronization Constructs : Barrier Directive

Purpose:

- It synchronizes all threads in the team
- When a barrier directive is reached, a thread will wait at that point until all other threads have reached that barrier.
- All threads then resume executing in parallel the code that follows the barrier.

```
#pragma omp barrier    newline
```

6.4 Synchronization Constructs : Taskwait Directive

Purpose:

- It specifies a wait on the completion of child tasks generated since the beginning of the current task

```
#pragma omp taskwait   newline
```

Restrictions:

- It may be placed only at a point where a base language statement is allowed.
- It may not be used in place of the statement following an if, while, do, switch, or label.

6.5 Synchronization Constructs : Atomic Directive

Purpose:

- It specifies that a specific memory location must be updated atomically, rather than letting multliple threads attempt to write to it.
- It provides a mini-critical section

```
#pragma omp atomic   newline
        statement_expression
```

Restrictions:

- It applies only to a single, immediately following statement
- An atomic statement must follow a specific syntax

6.6 Synchronization Constructs : Flush Directive

Purpose:

```
#pragma omp atomic   newline
        statement_expression
```