

- **Coordinate execution of multiple threads**
- **Used when**
  - **A desired execution order is required**
  - **Presence of atomicity of a set of instructions**
  - **Need for serial execution of code segments**

- **The barrier directive**

```
!$OMP barrier
```

```
#pragma omp barrier
```

- **All threads in a team wait until others have caught up**

```

#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
        for(i=0; i<N; i++) { C[i]=big_calc3(i, A); }
    #pragma omp for nowait
        for(i=0; i<N; i++) { B[i]=big_calc2(C, i); }
    A[id] = big_calc4(id);
}

```

implicit barrier at the end of for construct

no implicit barrier due to nowait

implicit barrier at end of a parallel region

```
!$OMP single [clause list]
```

```
... statements ...
```

```
!$OMP end single
```

```
#pragma omp single [clause list]
```

```
    structured block
```

- **Clause list:** `private`, `firstprivate`, and `nowait`
- **Only first thread that encounters single block enters it**
  - **Other threads proceed ahead (if `nowait` specified), otherwise wait at end of the single block**
- **Useful for computing global data and performing I/O**

```
!$OMP master
... statements ...
!$OMP end master
```

```
#pragma omp master
    structured block
```

- Only master thread executes the structured block
- No implicit barrier like the `single` directive

```
!$OMP critical [(name)]
... statements ...
!$OMP end critical
```

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

- **Used for implementing critical regions identified by name**
  - **Region that must be executed serially, one thread at a time**
- **No jumps permitted into or out of the structured block**

```
!$OMP atomic [(name)]
    statement
```

```
#pragma omp atomic [(name)]
    statement
```

- Implements critical section w.r.t. load and update of single memory location
- Update instruction can be `x binary_operation = expr, x++, ++x, x--, or -x`
  - `expr` should not include reference to `x`

```

#pragma omp parallel sections
{
    #pragma parallel section
    {
        /* producer thread */
        task = produce_task();
        #pragma omp critical(task_queue)
        {
            insert_into_queue(task);
        }
    }
    #pragma parallel section
    {
        /* consumer thread */
        #pragma omp critical(task_queue)
        {
            task = extract_from_queue(task);
        }
        consume_task(task);
    }
}

```

```
!$OMP ordered
... statements ...
!$OMP end ordered
```


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

- **Specifies in-order execution of a for loop**
  - **for or parallel for directive must also have ordered clause**
- **Only single thread can enter an ordered block when all prior threads (determined by loop indices) have exited**

```

cuml_sum[0] = list[0];
#pragma omp parallel for private(i) \
        shared(cumul_sum, list, n) ordered
for(i=1; i<n; i++)
{
    /* other processing on list[i] if needed */
    #pragma omp ordered
    {
        cumul_sum[i] = cumul_sum[i-1] + list[i];
    }
}

```



Parallel execution using ordered clause makes sense only if list[i] involves huge computation.

```
!$OMP flush [(list)]
```

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

- Forces shared variables in `list` to be written to or read from the memory system
  - Ensures memory consistency across threads

- **OpenMP directives that have implicit flush**
  - **barrier, entry and exit of critical, ordered, parallel, parallel for, parallel sections blocks, and at the exit of for, sections, and single blocks**
- **A flush is not implied**
  - **If nowait clause is present**
  - **At entry of for, sections, and single blocks**
  - **At entry or exit of a master block**

- `private`
  - When thread uses a variable that no other thread accesses
- `firstprivate`
  - When thread repeatedly uses a variable initialized earlier in the program
- `reductions`
  - When multiple threads keep local counts that need to be accrued at the end of parallel block
- `shared`
  - When above techniques are not applicable, remaining data items may be shared among various threads

- **Variables that persist across different parallel regions can be specified by threadprivate directive**

```
#pragma omp threadprivate(variable_list)
```

  - **Number of threads should be same across all parallel regions**
    - **Dynamic adjustment of no. of threads should be disabled**
  
- **copyin(variable\_list) assigns same value to threadprivate variables across all threads in a parallel region**

- Controlling Number of Threads and Processors

```
void omp_set_num_threads(int num_threads);
int omp_get_num_threads(); /* number of threads */
integer function omp_get_num_threads();
int omp_get_num_procs(); /* number of processors */
```

```
int omp_get_max_threads();
```

- Returns max. no. of threads that could be created by parallel that has no num\_threads clause

```
int omp_get_thread_num();
```

```
integer function omp_get_thread_num();
```

- Returns unique thread i.d. for each thread in a team

```
int omp_in_parallel();
```

- Returns non-zero value if called from within a parallel region

- **Controlling and Monitoring Thread Creation**

```
void omp_set_dynamic(int dynamic_threads);
```

- **Dynamically alter number of threads created in a parallel region**
- **If `dynamic_threads` is zero, dynamic adjustment is disabled**

```
int omp_get_dynamic();
```

- **Returns non-zero if dynamic adjustment is enabled, else zero**

```
void omp_set_nested(int nested);
```

- **If `nested` is zero, nested parallelism is disabled (nested parallel regions are then serialized)**

```
int omp_get_nested();
```

- **Returns non-zero if nested parallelism is enabled, else zero**

- **Mutual Exclusion**

- Includes functions for initializing, locking, unlocking, and discarding locks (in situations that require explicit locks)
- Lock data structure is of type `omp_lock_t`

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock); /* return(0)=in use */
```

- **OpenMP also supports nestable locks that can be locked multiple times by same thread**

- Functions for handling nested lock similar as above

```
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

...

- **OMP\_NUM\_THREADS**
  - **Specifies default number of threads in a parallel region**
    - **Can change no. of threads using `omp_set_num_threads` function or `num_threads` clause in a `parallel` directive**
  
- **OMP\_DYNAMIC**
  - **When TRUE, allows no. of threads to be controlled at runtime using `omp_set_num_threads/num_threads`**

- **OMP\_NESTED**
  - **When set to TRUE, enables nested parallelism**
    - **Unless it is disabled by calling `omp_set_nested` function with zero arguments**
  
- **OMP\_SCHEDULE**
  - **Controls assignment of iteration spaces in `for` directive having runtime scheduling class/type**
    - **Can take values `static`, `dynamic`, and `guided` along with chunk size (`setenv OMP_SCHEDULE "static 4"`)**

```

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

#define N 1000000
#define PI 3.1415926535
#define DELTA .01415926535

int main (int argc, char *argv[])
{
    int nthreads, tid, i;
    float a[N], b[N];
    omp_lock_t locka, lockb;

    /* Initialize the locks */
    omp_init_lock(&locka);
    omp_init_lock(&lockb);

```

```

/* Fork a team of threads giving them their own copies of
   variables */
#pragma omp parallel shared(a, b, nthreads, locka, lockb)
                    private(tid) num_threads(2)
{

    /* Obtain thread number and number of threads */
    tid = omp_get_thread_num();
    #pragma omp master
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);
    #pragma omp barrier

```

```

#pragma omp sections nowait
{
    #pragma omp section
    {
        printf("Thread %d initializing a[]\n", tid);
        omp_set_lock(&locka);
        for (i=0; i<N; i++)
            a[i] = i * DELTA;
        omp_unset_lock(&locka);
        omp_set_lock(&lockb);
        printf("Thread %d adding a[] to b[]\n", tid);
        for (i=0; i<N; i++)
            b[i] += a[i];
        omp_unset_lock(&lockb);
    }
}

```

```

#pragma omp section
{
    printf("Thread %d initializing b[]\n",tid);
    omp_set_lock(&lockb);
    for (i=0; i<N; i++)
        b[i] = i * PI;
    omp_unset_lock(&lockb);
    omp_set_lock(&locka);
    printf("Thread %d adding b[] to a[]\n",tid);
    for (i=0; i<N; i++)a[i] += b[i];
    omp_unset_lock(&locka);
}
} /* end of sections */
} /* end of parallel region */
} /* end of main */

```

```

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

#define NRA 62          /* number of rows in matrix A */
#define NCA 15         /* number of columns in matrix A */
#define NCB 7          /* number of columns in matrix B */

int main (int argc, char *argv[])
{
    int tid, nthreads, i, j, k, chunk;
    double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB];
    /* Matrix A and B to be multiplied, C is the resultant matrix */

    chunk = 10;        /* set loop iteration chunk size */

```

```

/* Spawn a parallel region explicitly scoping all variables */
#pragma omp parallel shared(a, b, c, nthreads, chunk)
    private(tid, i, j, k)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Starting matrix multiple example with
            %d threads \n", nthreads);
        printf("Initializing matrices...\n");
    }
}

```

```

/* Initialize matrices */
#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
        a[i][j] = i+j;

#pragma omp for schedule (static, chunk)
for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
        b[i][j] = i*j;

#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
        c[i][j] = 0;

```

```

/* Do matrix multiply sharing iterations on outer loop */
/* Display who does which iterations for demo purposes */
printf("Thread %d starting matrix multiply...\n",tid);

#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
{
    printf("Thread=%d did row=%d\n",tid,i);
    for(j=0; j<NCB; j++)
        for (k=0; k<NCA; k++)
            c[i][j] += a[i][k] * b[k][j];
}
} /* End of parallel region */

```

```

/** Print results */
printf("*****\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
{
    for (j=0; j<NCB; j++)
        printf("%6.2f  ", c[i][j]);
    printf("\n");
}
printf("*****\n");
printf ("Done.\n");
}

```

**Compile options:**

```

icc -openmp omp_matrix.c -o matrix
pathcc -mp omp_matrix.c -o matrix
pgcc -mp omp_matrix.c -o matrix
gcc -fopenmp omp_matrix.c -o matrix

```