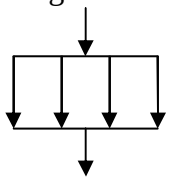
OpenMP
E.g.

```
setenv OMP_NUM_THREADS 8

#include <omp.h>
void main(void)
{
        int Myid, Nprocs;
        #pragma omp parallel private(Myid, Nprocs) num_threads(5)
        {
                Myid = omp_get_thread_num();
                Nprocs = omp_get_num_thread();
                printf("\n I am %d of %d \n", Myid, Nprocs);
        } ← There is a barrier
}
```

OpenMP

- API to multiple threading
- Intention: have programs that execute correctly in both serial/parallel
- Programming model: fork/join



Program in OMP

Insert pragmas

```
#pragma omp <name>[clause]
```

To start a parallel computation:

```
#pragma omp parallel [clause]
{
        :
}
```

Clauses

num_threads(n)
private(Myid)
shared(i, j)

Executing a loop

```
#pragma omp parallel for
for(i=1; i<n; i++)
        b[i] = a[i]+c[i];
```

At the end of the parallel region, a barrier is implied. To avoid this implied barrier, one can use the nowait directive:

```
#pragma omp parallel
```

```
{
    #pragma omp for nowait
    for(i=1; i<n; i++){…}
}
```

Critical Regions

```
#pragma omp critical(name)
{
}
```

E.g. deque:

```
#pragma omp parallel shared(x)
{
    #pragma omp critical(x-queue)
    x_next = deque(x);
}
```

Reduction operation

E.g. Each threads computes a partial sum, and the master thread gets the sum of these partial results

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

Parallel Sections

```
#pragma omp parallel sections
{
    #pragma omp section
    { #1 }
    #pragma omp section
    { #2 }
    #pragma omp section
    { #3 }
}
```

Single:

```
#pragma omp parallel
{
    #pragma omp single nowait
    printf("\n Begin work \n");
}
```

Ordered Sections

Sequentially ordered output

```
#pragma omp for ordered
```

```
for(i=0; i<n; i++)
        printf(\n %d", i);
```

To set the number of thread to a fixed value:

```
omp_set_dynamic(0);
omp_set_num_threads(16);
```

To avoid race conditions: atomic

E.g.

```
#pragma omp parallel for shared(x, y, index, n)
for(i=0; i<n; i++)
{
        #pragma omp atomic(x)
        x[index[i]] += work1(i);
        y[i] += work2(i);
}
```

The flush:

```
#pragma omp parallel
{
        #pragma omp flush ← consistent view of all shared objects
}
```

Get number of threads:

```
#pragma omp parallel
{
        n = omp_get_num_threads();
        I = omp_get_threads_no();
}
```

Locks:

```
#include <omp.h>
int main()
{
        omp_lock_t lck;
        omp_init_lock(&lck);
        #pragma omp parallel shared(lck)
        {
                omp_set_lock(&lck);
                        :
                omp_unset_lock(&lck);
        }
}
omp_test_lock(&lck)
omp_destroy_lock(&lck)
```

Nested for directives

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

The barrier directive:

```
#pragma omp barrier

void w5(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for(i=0; i<n; i++)
        {
            work1(i);
            #pragma omp barrier        ⇐ BAD!
            work2(i);
        }
}

void w6(…)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp barrier        ⇐ BAD!
        }
        finish();
    }
}
```

Scoping variables

```
int i = 1;
#pragma omp parallel lastprivate(i)
{
    i=3;
}
```

Matrix Multiplication

```
#pragma omp parallel shared (a, b, c)
{
        for(i=0; i<n; i++){
                for(j=0; j<n; j++){
                        for(k=0; k<n; k++){
                        }
                }
        }
}
```