

OpenMP

# What is OpenMP?

- A collection of **compiler directives**, **library routines**, and **environment variables** that can be used to specify shared memory parallelism.
- Designed with the cooperation of many computer vendors including Intel, HP, IBM, and SGI. For this reason it has become the standard (and therefore portable) way of programming SMPs.
- The Fortran directives are very similar to the C/C++ OpenMP directives.

# The PARALLEL directive

- The **parallel** directive defines a parallel region and constitutes as parallel construct.
- An OpenMP program begins execution as a single task, called the master thread. When a **parallel** construct is encountered, the master thread creates a team of threads. The statements enclosed by the parallel construct, including routines called from within the enclosed construct, are executed in parallel by each thread in the team.
- At the end of the parallel construct the threads in the team synchronize and only the master thread continues execution.
- The general form of this construct is:  
***#pragma omp parallel [parallel-clause ...] newline***  
*parallel region*
- There are several classes of parallel-clauses. Next, we discuss the private(list)clause.

# Parallel Directive clause

- By default, all variables are shared by all tasks in parallel region
- If **private** clause is used
  - Each thread will receive a separate copy listed in clause
  - There will be an additional copy of the variable that can be accessed outside the parallel region
- Undefined behavior for variables defined as **private**:
  - Upon entering the construct
  - Upon exit from the construct
- Other clauses available (firstprivate, shared, reduction, etc..)

# Parallel directive example

- As an example, consider the following **sequential** code segment

```
c = sin (d);  
for (i=1; i < n; i++)  
    a[i] = b[i] + c;  
e = a[20]+ a[15];
```

- A simple OpenMP implementation would take the form:

```
c = sin(d);  
#pragma omp parallel private (i, il, iu)  
{  
    get_limits(n,&il,&iu,omp_get_num_threads(), omp_get_thread_num());  
    for (i=il; i < iu; i++)  
        a[i] = b[i] + c;  
}  
e = a[20] + a[15];
```

# get\_limits()

- User supplied routine to determine how much work each threads to do
  - i.e. Starting and ending points to iterate to

```
void get_limits(int n, int *il, int *lu, int nthreads, int tnum) {  
    /* Figure out chunk size each thread needs */  
    int chunksize = n / nthreads;  
  
    /* Starting iteration point for the thread  
    *il = tnum * chunksize;  
  
    /* Ending iteration point for the thread – (tnum + 1)*chunksize  
    *iu = *il + chunksize;  
}
```

# Private directive example (cont)

- Notice that the first statement can be incorporated into the parallel region. In fact, **c** can be declared as private assuming it is never used outside the loop.

```
#pragma omp parallel private (c, i, il, iu)
{
    c = sin(d);
    get_limits(n,&il,&iu, omp_get_num_threads(), omp_get_thread_num());
    for (i=il; i < iu; i++)
        a[i] = b[i] + c;
}
e = a[20] + a[15];
```

# Barrier directive

- To incorporate **e** into the parallel region it is necessary to make sure that **a[20]** and **a[15]** have been computed before the statement executes.
- This can be done with a **barrier directive**
  - Synchronizes all the threads in the parallel region (at that barrier point)
  - Each thread waits until all the others in the team have reached the barrier

```
#pragma omp parallel private (c, i, il, iu)
{
    c = sin(d);
    get_limits(n,&il,&iu, omp_get_num_threads(), omp_get_thread_num());
    for (i=il; i < iu; i++)
        a[i] = b[i] + c;
    #pragma omp barrier
    e = a[20] + a[15];
}
```

# SINGLE directive

Since **e** is shared:

- Redundant for all threads to assign to **e** (in this example)
  - Bad performance due to all threads competing for access to single location
  - Only one task needs to execute the assignment
- This can be accomplished with the **single** directive:

```
#pragma omp parallel private (c, i, il, iu)
{
    c = sin(d);
    get_limits(n,&il,&iu, omp_get_num_threads(), omp_get_thread_num());
    for (i=il; i < iu; i++)
        a[i] = b[i] + c;
    #pragma omp barrier
    #pragma omp single
        e = a[20] + a[15];
}
```

# Single directive syntax

```
#pragma omp single [single-clause [single-clause] ...]  
    block
```

- This directive specifies that the enclosed region of code is to be executed by one and only one of the tasks in the team.
- Tasks in the team not executing the single block wait at the end of single, unless `nowait` is specified. In this case, there is no need for this implicit barrier since one already exists at the end parallel directive.
- One of the two single-clauses is `private(list)`.

# Single example #2

```
void sp_1a(a,b,n) {  
#pragma omp parallel private(i)  
{  
    #pragma omp for  
    for (i=1; i < n; i++)  
        a[i]=1.0/a[i];  
    #pragma omp single  
        a[1]=min(a[1],1.0);  
    #pragma omp for nowait  
    for (i=1; i < n; i++)  
        b[i]=b[i]/a[i];  
}  
}
```

# FOR directive

Simpler way to write a parallel loop

- No more programmer explicit computation of loop bounds

```
#pragma omp parallel private (c, i)
{
    c = sin(d);
    #pragma omp for schedule (static)
    for (i=1; i < n; i++)
        a[i] = b[i] + c;
    #pragma omp single nowait
        e = a[20] + a[15];
}
```

# FOR directive (cont.)

The syntax of the **for** directive is as follows:

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

There are several for clauses including private, schedule, and nowait.

The schedule could assume other values including dynamic.

The nowait clause eliminates the implicit barrier at the end of the directive. In the previous example, the nowait clause should not be used.

# FOR example

- An example of **for** with the **nowait** directive:

```
for_2(a,b,c,d,m,n) {
    float a[n][n], b[n][n], c[m][m], d[m][m];
    #pragma omp parallel private(i,j)
    {
        #pragma for schedule(dynamic) nowait
        for (i=2; i < n; i++)
            for (j = 1; j < i; j++)
                b[j][i]=(a[j][i]+a[j][i+1])/2;
        #pragma for schedule (dynamic) nowait
        for (i=2; i < m; i++)
            for (j = 1; j < i; j++)
                d[i][j]=(c[j][i]+c[j][i-1])/2
    }
}
```

# Parallel for directive

- An alternative to the for is the **parallel for** directive which is no more than a shortcut for a parallel directive containing a single for directive.

For example, the following code segment:

```
#pragma omp parallel private(i)
{
    #pragma for schedule(dynamic) nowait
    for (i = 1; i < n; i++)
        b[i]=(a[i]+a[i+1])/2;
}
```

could be rewritten as

```
#pragma omp parallel for private(i) schedule(dynamic)
    for (i = 1; i < n; i++)
        b[i]=(a[i]+a[i+1])/2;
```

The omp parallel for is an example of **worksharing**

# Parallel for example

The previous routine for\_2 can be rewritten as follows:

```
for_2(a,b,c,d,m,n) {  
    float a[n][n], b[n][n], c[m][m], d[m][m];  
    #pragma omp parallel for private(i,j) schedule(dynamic)  
    {  
        for (i=2; i < n; i++)  
            for (j = 1; j < i; j++)  
                b[j][i]=(a[j][i]+a[j][i+1])/2;  
    }  
    #pragma omp parallel for private(i,j) schedule(dynamic)  
    {  
        for (i=2; i < m; i++)  
            for (j = 1; j < i; j++)  
                d[i][j]=(c[j][i]+c[j][i-1])/2  
    }  
}
```

# Disadvantages to previous example

There are two disadvantages to this last version of for\_2:

1. There is a barrier at the end of the first loop.
2. There are two parallel regions. There is overhead at the beginning of each.

# Sections directive

An alternative way to write the for\_2 routine is:

```
for_2(a,b,c,d,m,n) {  
    float a[n][n], b[n][n], c[m][m], d[m][m];  
    #pragma omp parallel private(i,j)  
    {  
        #pragma omp sections nowait  
        {  
            #pragma omp section  
            for (i=2; i < n; i++)  
                for (j = 1; j < i; j++)  
                    b[j][i]=(a[j][i]+a[j][i+1])/2;  
            #pragma omp section  
            for (i=2; i < m; i++)  
                for (j = 1; j < i; j++)  
                    d[i][j]=(c[j][i]+c[j][i-1])/2  
        }  
    }  
}
```

# Sections directive (cont.)

The **sections** directive specifies that the enclosed sections of code are to be divided among threads in the team. Each **section** is executed by one thread in the team. Its syntax is as follows:

```
#pragma omp sections[sections-clause [sections-clause] ...]
```

```
#pragma omp section
```

```
  block
```

```
[#pragma omp section
```

```
  block
```

```
  .
```

```
  .
```

```
.]
```

# Critical regions and Reductions

Consider the following loop:

```
for (i=1;i<n;i++) {  
    for (j=1; j < m; j++) {  
        ia[i][j]=b[i][j]+d[i][j];  
        isum=isum+ia[i][j];  
    }  
}
```

Here, we have a race due to `isum`. This race cannot be easily removed.

However, the `+` operation used to compute `isum` is associative and `isum` only appears in the statement that computes its value.

The integer addition operation is not really associative, but in practice we can assume it is if the numbers are small enough so there is never any overflow.

# Critical directive

Under these circumstances, the loop can be transformed into the following form:

```
#pragma omp parallel private(local_isum)
{
    local_isum=0;
    #pragma for nowait
    {
        for (i=1; i < n; i++)
        {
            for (j=1; j < m; j++) {
                local_isum=local_isum + ia[i][j];
                #pragma omp critical
                isum = isum + local_isum;
            }
        }
    }
}
```

# Critical directive (cont.)

We use the **critical directive** to avoid the following problem.

The statement

```
    isum=isum+local_isum
```

will be translated into a machine language sequence similar to the following:

```
load    register_1,isum
load    register_2,local_isum
add     register_3,register_1,register_2
store   register_3,isum
```

# Critical directive (cont.)

Assume now there are two tasks executing the statement

***isum=isum+local\_isum***

simultaneously.

In one **local\_sum** is 10, and in the other 15. Assume isum is 0 when both tasks start executing the statement.

Time	Task 1	isum	Task 2
1	load r1,local_isum	0	
2	load r2, isum	0	load r1, local_isum
3	add r3, r2, r1	0	load r2, isum
4	store r3, isum	10	add r3, r2, r1
5		15	store r3, isum

As can be seen, interleaving the instructions between the two tasks produces incorrect results. The **critical** directive precludes this interleaving. Only one task at a time can execute a critical region with the same name.

# Critical directive (cont.)

The assumption is that it does not matter in which order the tasks enter a critical region as long as they are never inside a critical region of the same name at the same time.

# Reduction

An alternative way of writing the above parallel loop is:

```
#pragma omp parallel for reduction(+:isum)
```

```
  for (i=1; i < n; i++)
```

```
    for (j = 1; j < n; j++)
```

```
      isum=isum+ia(j,i)
```

The reduction clause can be applied to a number of operations.

# Tasking in OpenMP 3.0

- Biggest addition in OpenMP 3.0 spec
- Allows to parallelize irregular problems
  - Unbounded loops
  - Recursive algorithms
  - Etc..
- Tasks are work units which may be deferred (or not)
- Tasks executed by threads in the team
- Each task encountered by thread is created

# Tasking example

```
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single nowait
    {
        #pragma omp task
        bar();
    }
    moo();
}
```

# Tasking example #2

Multiple list traversal:

List l[N]

```
#pragma omp parallel
```

```
{
```

```
    #pragma omp for
```

```
    for (int i = 0; i < N; i++) {
```

```
        Element e;
```

```
        for ( e = l[i]->first; e ; e = e->next ) {
```

```
            #pragma omp task
```

```
                process(e);
```

```
        }
```

```
    }
```

```
}
```

# Tasking example #3 (Fibonacci)

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
        x = fib(n-1);
    #pragma omp task shared(y)
        y = fib(n-2);
    #pragma omp taskwait
        return x+y;
}
```

# Task directive

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

Each encountering thread creates a new task

- Packages code and data

Tasks can be nested:

- Within a task
- Within a worksharing construct

**#pragma omp taskwait**

- Waits for all child tasks to complete

# Compiling OpenMP

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

int main(void) {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        printf("Hello World\n");
    }
}
```

To compile using gcc:

**gcc -fopenmp test.c -o test**

To compile using icc:

**icc -openmp test.c -o test**

# OpenMP Performance

- Performance of a parallel matrix-matrix multiplication
  - (1000x1000)
  - Using blocking
- **Red** bar represents performance (MFLOPS) for a particular number of threads
- Compiled in gcc:
  - `gcc -O2 -fopenmp mm.c`

