Aim: To get familiarized with OpenMP Directives

OpenMP Directives

Provides links to directives used in the OpenMP API.
Visual C++ supports the following OpenMP directives.
For parallel work-sharing:

TABLE 1

Directive	Description
parallel	Defines a parallel region, which is code that will be executed by multiple threads in parallel.
for	Causes the work done in a for loop inside a parallel region to be divided among threads.
sections	Identifies code sections to be divided among all threads.
single	Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.

For master and synchronization:

TABLE 2

Directive	Description
master	Specifies that only the master thread should execute a section of the program.
critical	Specifies that code is only executed on one thread at a time.
barrier	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.
atomic	Specifies that a memory location that will be updated atomically.
flush	Specifies that all threads have the same view of memory for all shared objects.
ordered	Specifies that code under a parallelized for loop should be executed like a sequential loop.

For data environment:

TABLE 3

Directive	Description
threadprivate	Specifies that a variable is private to a thread.

atomic

Specifies that a memory location that will be updated atomically.
C++Copy

```
#pragma omp atomic
    expression
```

Parameters

expression
The statement that has the lvalue, whose memory location you want to protect against more than one write.

Remarks

The atomic directive supports no clauses.
For more information, see 2.6.4 atomic construct.

barrier

Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.

C++Copy

```
#pragma omp barrier
```

Remarks

The barrier directive supports no clauses.
For more information, see 2.6.3 barrier directive.

critical

Specifies that code is only be executed on one thread at a time.

C++Copy

```
#pragma omp critical [(name)]
{
    code_block
}
```

Parameters

name
(Optional) A name to identify the critical code. The name must be enclosed in parentheses.

Remarks

The critical directive supports no clauses.
For more information, see 2.6.2 critical construct.

flush

Specifies that all threads have the same view of memory for all shared objects.
C++Copy
#pragma omp flush [(var)]

Parameters

var
(Optional) A comma-separated list of variables that represent objects you want to synchronize. If var isn't specified, all memory is flushed.

Remarks

The flush directive supports no clauses.
For more information, see 2.6.5 flush directive.

for

Causes the work done in a for loop inside a parallel region to be divided among threads.
C++Copy
#pragma omp [parallel] for [clauses]
    for_statement

Parameters

clauses
(Optional) Zero or more clauses, see the **Remarks** section.
for_statement
A for loop. Undefined behavior will result if user code in the for loop changes the index variable.

Remarks

The for directive supports the following clauses:

- private
- firstprivate
- lastprivate

- •reduction
- •ordered
- •schedule
- •nowait

If parallel is also specified, clauses can be any clause accepted by the parallel or for directives, except nowait.
For more information, see 2.4.1 for construct.

master

Specifies that only the master thread should execute a section of the program.

C++Copy

```
#pragma omp master
{
    code_block
}
```

Remarks

The master directive supports no clauses.
The single directive lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.
For more information, see 2.6.1 master construct.

ordered

Specifies that code under a parallelized for loop should be executed like a sequential loop.

C++Copy

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

Remarks

The ordered directive must be within the dynamic extent of a for or parallel for construct with an ordered clause.
The ordered directive supports no clauses.
For more information, see 2.6.6 ordered construct.

parallel

Defines a parallel region, which is code that will be executed by multiple threads in parallel.

C++Copy

```
#pragma omp parallel [clauses]
{
    code_block
}
```

Parameters

clauses
(Optional) Zero or more clauses, see the **Remarks** section.

Remarks

The parallel directive supports the following clauses:

- •if
- •private
- •firstprivate
- •default
- •shared
- •copyin
- •reduction
- •num_threads

parallel can also be used with the for and sections directives.
For more information, see 2.3 parallel construct.

sections

Identifies code sections to be divided among all threads.

C++Copy

```
#pragma omp [parallel] sections [clauses]
{
    #pragma omp section
    {
        code_block
    }
}
```

Parameters

clauses
(Optional) Zero or more clauses, see the **Remarks** section.

Remarks

The sections directive can contain zero or more section directives.
The sections directive supports the following clauses:

- •private
- •firstprivate
- •lastprivate
- •reduction
- •nowait

If parallel is also specified, clauses can be any clause accepted by the parallel or sections directives, except nowait.
For more information, see 2.4.2 sections construct.

single

Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.

C++Copy

```
#pragma omp single [clauses]
{
    code_block
}
```

Parameters

clauses
(Optional) Zero or more clauses, see the **Remarks** section.

Remarks

The single directive supports the following clauses:

- •private
- •firstprivate
- •copyprivate
- •nowait

The master directive lets you specify that a section of code should be executed only on the master thread.
For more information, see 2.4.3 single construct.

threadprivate

Specifies that a variable is private to a thread.

C++Copy

```
#pragma omp threadprivate(var)
```

Parameters

var
A comma-separated list of variables that you want to make private to a thread. var must be either a global- or namespace-scoped variable or a local static variable.

Remarks

The threadprivate directive supports no clauses.
The threadprivate directive is based on the thread attribute using the __declspec keyword; limits on __declspec(thread) apply to threadprivate. For example, a threadprivate variable will exist in any thread started in the process, not just those threads that are part of a thread team spawned by a parallel region. Be aware of this implementation detail; you may notice that constructors for a threadprivate user-defined type are called more often then expected.
You can use threadprivate in a DLL that is statically loaded at process startup, however you can't use threadprivate in any DLL that will be loaded via LoadLibrary such as DLLs that are loaded with /DELAYLOAD (delay load import), which also uses LoadLibrary.
A threadprivate variable of a destructible type isn't guaranteed to have its destructor called. For example:

C++Copy

```
struct MyType
{
    ~MyType();
};

MyType threaded_var;
#pragma omp threadprivate(threaded_var)
int main()
{
    #pragma omp parallel
    {}
}
```

Users have no control as to when the threads constituting the parallel region will terminate. If those threads exist when the process exits, the threads won't be notified about the process exit, and the destructor won't be called

for threaded_var on any thread except the one that exits (here, the primary thread). So code shouldn't count on proper destruction of threadprivate variables.

Code:
Omp_loop.c :

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 5
#define N 30
int main (int argc, char *argv[])
{
int nthreads, tid, 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 shared(a,b,c,nthreads,chunk) private(i,tid)
{
tid = omp_get_thread_num();
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
printf("Thread %d starting...\n",tid);
#pragma omp for schedule(dynamic,chunk)
for (i=0; i<N; i++)
{
c[i] = a[i] + b[i];
printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
} /* end of parallel section */
}
```

Omp_sect.c :

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 19

int main (int argc, char *argv[])
{
int i, nthreads, tid;
float a[N], b[N], c[N], d[N];
/* Some initializations */
for (i=0; i<N; i++) {
a[i] = i * 1.5;
b[i] = i + 22.35;
c[i] = d[i] = 0.0;
}
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
```

```
{
tid = omp_get_thread_num();
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
printf("Thread %d starting...\n",tid);
#pragma omp sections nowait
{
#pragma omp section
{
printf("Thread %d doing section 1\n",tid);
for (i=0; i<N; i++)
{
c[i] = a[i] + b[i];
printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
}
#pragma omp section
{
printf("Thread %d doing section 2\n",tid);
for (i=0; i<N; i++)
{
d[i] = a[i] * b[i];
printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
}
}

} /* end of sections */
printf("Thread %d done.\n",tid);
} /* end of parallel section */
}
```

Oomp_matrix.c :

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define NRA 20 /* number of rows in matrix A */
#define NCA 10 /* 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], /* matrix A to be multiplied */
b[NCA][NCB], /* matrix B to be multiplied */
c[NRA][NCB]; /* result matrix C */
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 demonstration 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");
}
```

Output:

Omp_loop.c :

```
151070031@ltsp184:~/Desktop/pca$ ./omp_loop
Number of threads = 4
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 1 starting...
Thread 1: c[5]= 10.000000
Thread 1: c[6]= 12.000000
Thread 1: c[7]= 14.000000
Thread 1: c[8]= 16.000000
Thread 1: c[9]= 18.000000
Thread 1: c[10]= 20.000000
Thread 3 starting...
Thread 3: c[15]= 30.000000
Thread 3: c[16]= 32.000000
Thread 3: c[17]= 34.000000
Thread 1: c[11]= 22.000000
Thread 1: c[12]= 24.000000
Thread 1: c[13]= 26.000000
Thread 1: c[14]= 28.000000
Thread 1: c[20]= 40.000000
Thread 1: c[21]= 42.000000
Thread 1: c[22]= 44.000000
Thread 1: c[23]= 46.000000
Thread 1: c[24]= 48.000000
Thread 1: c[25]= 50.000000
Thread 1: c[26]= 52.000000
Thread 1: c[27]= 54.000000
Thread 1: c[28]= 56.000000
Thread 1: c[29]= 58.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 3: c[18]= 36.000000
Thread 3: c[19]= 38.000000
Thread 2 starting...
151070031@ltsp184:~/Desktop/pca$
```

Omp_sect.c :

```
151070031@ltsp184:~/Desktop/pca$ ./omp_se
Number of threads = 4
Thread 0 starting...
Thread 0 doing section 1
Thread 0: c[0]= 22.350000
Thread 0: c[1]= 24.850000
Thread 2 starting...
Thread 2 doing section 2
Thread 2: d[0]= 0.000000
Thread 2: d[1]= 35.025002
Thread 2: d[2]= 73.050003
Thread 2: d[3]= 114.075005
Thread 2: d[4]= 158.100006
Thread 2: d[5]= 205.125000
Thread 2: d[6]= 255.150009
Thread 2: d[7]= 308.175018
Thread 2: d[8]= 364.200012
Thread 2: d[9]= 423.225006
Thread 2: d[10]= 485.249969
Thread 2: d[11]= 550.274963
Thread 2: d[12]= 618.299988
Thread 2: d[13]= 689.324951
Thread 2: d[14]= 763.349976
Thread 2: d[15]= 840.374939
Thread 2: d[16]= 920.399963
Thread 2: d[17]= 1003.424988
Thread 2: d[18]= 1089.449951
Thread 2 done.
Thread 3 starting...
Thread 3 done.
Thread 1 starting...
Thread 1 done.
Thread 0: c[2]= 27.350000
Thread 0: c[3]= 29.850000
Thread 0: c[4]= 32.349998
Thread 0: c[5]= 34.849998
Thread 0: c[6]= 37.349998
Thread 0: c[7]= 39.849998
Thread 0: c[8]= 42.349998
Thread 0: c[9]= 44.849998
Thread 0: c[10]= 47.349998
Thread 0: c[11]= 49.849998
Thread 0: c[12]= 52.349998
Thread 0: c[13]= 54.849998
Thread 0: c[14]= 57.349998
Thread 0: c[15]= 59.849998
Thread 0: c[16]= 62.349998
Thread 0: c[17]= 64.849998
Thread 0: c[18]= 67.349998
Thread 0 done.
151070031@ltsp184:~/Desktop/pca$
```

Oomp_matrix.c :

```
151070031@ltsp184:~/Desktop/pca$ ./omp_matrix
Starting matrix multiple example with 4 threads
Initializing matrices...
Thread 0 starting matrix multiply...
Thread=0 did row=0
Thread=0 did row=1
Thread=0 did row=2
Thread=0 did row=3
Thread=0 did row=4
Thread 3 starting matrix multiply...
Thread 1 starting matrix multiply...
Thread=1 did row=10
Thread=1 did row=11
Thread=1 did row=12
Thread=1 did row=13
Thread=1 did row=14
Thread=1 did row=15
Thread=1 did row=16
Thread=1 did row=17
Thread=1 did row=18
Thread=1 did row=19
Thread=0 did row=5
Thread=0 did row=6
Thread=0 did row=7
Thread=0 did row=8
Thread=0 did row=9
Thread 2 starting matrix multiply...
*******************************************************
Result Matrix:
  0.00   285.00   570.00   855.00   1140.00   1425.00   1710.00
  0.00   330.00   660.00   990.00   1320.00   1650.00   1980.00
  0.00   375.00   750.00   1125.00   1500.00   1875.00   2250.00
  0.00   420.00   840.00   1260.00   1680.00   2100.00   2520.00
  0.00   465.00   930.00   1395.00   1860.00   2325.00   2790.00
  0.00   510.00   1020.00   1530.00   2040.00   2550.00   3060.00
  0.00   555.00   1110.00   1665.00   2220.00   2775.00   3330.00
  0.00   600.00   1200.00   1800.00   2400.00   3000.00   3600.00
  0.00   645.00   1290.00   1935.00   2580.00   3225.00   3870.00
  0.00   690.00   1380.00   2070.00   2760.00   3450.00   4140.00
  0.00   735.00   1470.00   2205.00   2940.00   3675.00   4410.00
  0.00   780.00   1560.00   2340.00   3120.00   3900.00   4680.00
  0.00   825.00   1650.00   2475.00   3300.00   4125.00   4950.00
  0.00   870.00   1740.00   2610.00   3480.00   4350.00   5220.00
  0.00   915.00   1830.00   2745.00   3660.00   4575.00   5490.00
  0.00   960.00   1920.00   2880.00   3840.00   4800.00   5760.00
  0.00   1005.00   2010.00   3015.00   4020.00   5025.00   6030.00
  0.00   1050.00   2100.00   3150.00   4200.00   5250.00   6300.00
  0.00   1095.00   2190.00   3285.00   4380.00   5475.00   6570.00
  0.00   1140.00   2280.00   3420.00   4560.00   5700.00   6840.00
*******************************************************
Done.
```

Conclusion:
Thus,
1. Various work-sharing constructs were studied in detail.
2. OpenMP directives and Clauses are now known.