Example #1

- **What is wrong in the pseudo code below assuming that the code wants to call the function work with variable *i* where *i* goes from 0 to *np*-1? And how would you correct that?**

```
np = omp_get_num_threads() ;
#pragma omp parallel for schedule(static)
  for (i=0; i<np; i++)
    work(i);
```

Example #1 Solution

- **The OpenMP functions *omp_get_num_threads()* and *omp_get_thread_num()* are valid inside a parallel region only. Must re-write as:**

```
#pragma omp parallel private(i)
{
   i = omp_get_thread_num() ;
   work(i);
}
```

Example #2

- **What's the problem with this code fragment?**

```
 I=10
!$OMP PARALLEL
!$OMP& PRIVATE(I)
 I = I + 1
!$ OMP END PARALLEL
 PRINT*I
```

Example #2 Solution

- **I is undefined at this point. Need to rewrite:**

```
!$OMP PARALLEL
!$OMP& FIRSTPRIVATE(I)
 I = I + 1
!$ OMP END PARALLEL
 PRINT*I
```

Example #3

- **Parallelize the following serial program *without* using the reduction directive.**

```
#include <stdio.h>
main()
{
 int i, k = 0;
 for (i = 1; i <= 1000; i++)
    k += 1 ;
 printf("%d\n", k);
}
```

Example #3 Solution

```
#include <stdio.h>
main()
{
  int i, k = 0, k1;
  #pragma omp parallel shared(k) private(i,k1)
  {
   k1 = 0 ;
   #pragma omp for
   for (i=1; i <= 1000; i++)
      k1 += 1;
   #pragma omp critical
   k += k1;
   }
  printf("%d\n", k);
}
```

Example #4 Part 1

- **What will be the printed value of j in the following program:**

```
#include <stdio.h>
int j;
#pragma omp threadprivate(j)
int main()
{
  j = 1;
  #pragma omp parallel copyin(j)
  {
   #pragma omp master
   j = 2;
  }
  printf("j= %d\n", j);
}
```

Example #4 Part 1 Solution

- **The value 2 is printed by the master thread. Within a `master` section the master thread's copy of threadprivate variable is accessed and set to 2.**

Example #4 Part 2

- **Will the result of the above code be affected if the 'master' directive is changed to 'single' and how?**

Example #4 Solution Part 2

- **Here the printed value is indeterminate. In the `single` section, some single thread, but not necessarily the master thread, would set j to 2. The printing is always done by the master thread.**

Example #5

- **What's the problem with the following code:**

```
!$OMP PARALLEL
!$OMP& PRIVATE(I)
  I = I + 1
!$ OMP END PARALLEL
  Call Sub(I)
```

Example #5 Solution

- **I is undefined when the Sub(I) is called because it was declared *private*. It must be declared *lastprivate* to persist.**

```
!$OMP PARALLEL
!$OMP& LASTPRIVATE(I)
  I = I + 1
!$ OMP END PARALLEL
  Call Sub(I)
```

Example #6

- **What kind of scheduling will you suggest for the following and why?**

```
void sub_6(float a[], float b[], int n)
{
  int i, j ;
  #pragma omp parallel shared(a,b,n)
  private(i,j)
  {
   #pragma omp for schedule ( ???? ) nowait
   for (i = 1; i < n; i++)
    {
     for (j = 0; j <= i; j++)
       b[j+ n*i]=(a[j+n*i] + a[j+n*(i-1)])/2.;
    }
  }
}
```

Example #6 Solution

- **Since amount of work in each iteration is different we would use `schedule(dynamic, 1)` to get good load balance.**

Example #7

- **What's the problem in the following code:**

```
!$OMP PARALLEL ( Default )
!$OMP CRITICAL
  Call Sub(N)
!$OMP BARRIER
!$ OMP END CRITICAL
  Call Sub(2)
!$ OMP PARALLEL
```

Example #7 Solution

- **Problem: deadlock!**

- **Note on scheduling: Correct result should *not* depend on scheduling. Your code should work correctly with any scheduling. Run it with different scheduling to find the best performance.**

Example #8

- **Among the following two codes (sub8_a and sub8_b) , which one has more OpenMP overhead and why?**

```
void sub8_a(float a[],float b[],int n, int m)
{
  int i, j;
  for (j = 0; j < m ; j++)
   {
  #pragma omp parallel shared(a,b,n,j)
  private(i)
    {
     #pragma omp for nowait
     for (i = 0; i < n; i++)
      a[i + n*j] = b[i + n*j] /a[i + n*(j-1)];
    }
   }
}
```

Example #8 (continued)

```
void sub8_b(float a[],float b[],int n,int m)
{
  int i, j;
  #pragma omp parallel shared(a,b,n,m)
  private(i,j)
  {
  for (j = 0; j < m ; j++)
    {
     #pragma omp for nowait
     for (i = 0; i < n; i++)
       a[i + n*j] = b[i + n*j]/a[i + n*(j-1)];
    }
  }
}
```

Example #8 Solution

- **Since the `omp parallel` directive is outside the `j loop` , the `sub8_b` forms teams of threads less often, and thus reduces overhead.**

Example #9

- **What will Value of Z be once the above code is finished executing?**

```
INTEGER X(3), Y(3), Z
!$OMP PARALLEL DODEFAULT(PRIVATE) SHARED(X)
!$OMP REDUCTION( +: Z )
  DO K=1,3
  X(K) = K
  Y(K) = K*K
  Z = Z + X( K ) * Y( K )
  END DO
!$OMP END PARALLEL DO
```

Example #9 Solution

Variable	Thread 0	Thread 1	Thread 2
X	1	2	3
Y	1	4	9
Z	1	8	27

Z = 36 after loop finishes

Review

- **OpenMP is**
  - Portable, Shared Memory Multi-processing API
    - Fortran 77, Fortran 90, C, and C++
    - OS-neutral
  - Standardizes Fine-grained Parallelism
    - Supports Coarse-grained Algorithms
  - Directive-based
    - Single-source code solution

Parallelism in OpenMP

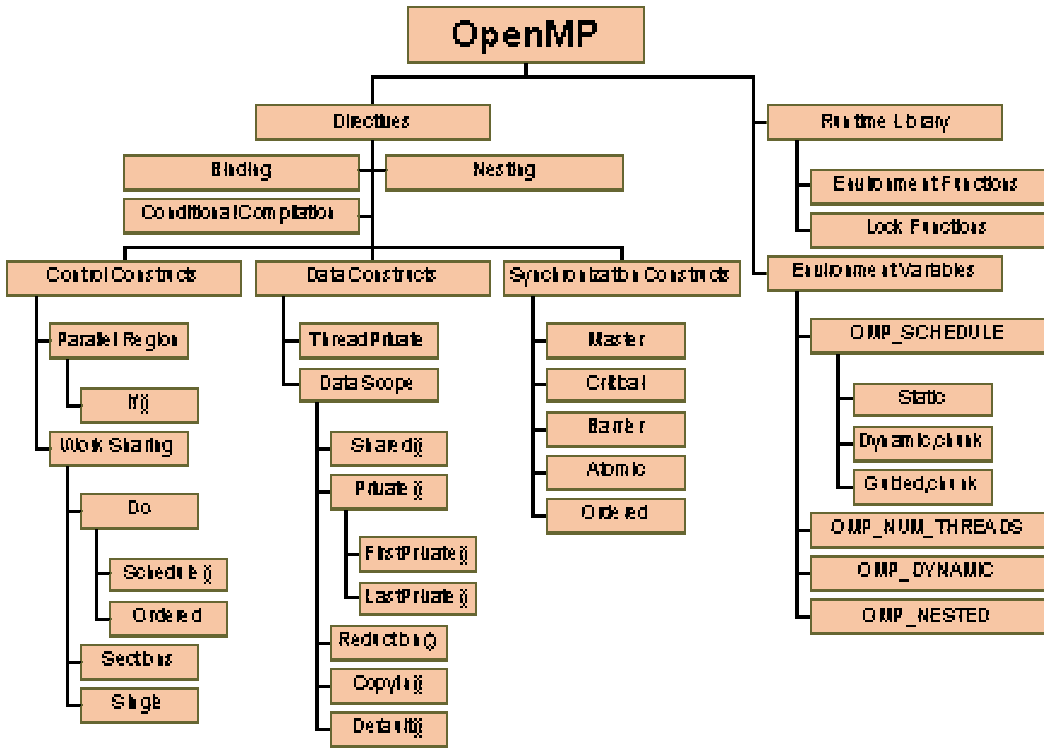
- **Parellelism achieved by threads and implented using fork-join model**
  - less flexible than Pthreads
  - much easier to write and maintain
  - offers most of the performance
- **Threads are created when first PARALLEL region is encountered**
- **Threads sleep (minimum impact on system resources) or spin (maximum performance) between PARALLEL regions**
  - *Logically*, just one thread between regions

Parallelism in OpenMP

- ***Only* have multiple threads in PARALLEL regions**
  - Marked by $OMP PARALLEL in Fortran, valid until $OMP END PARALLEL
  - Marked by #pragma omp parallel in C, valid only for next statement (or composite statement)
- ***Only* have parallelism inside work sharing constructs within PARALLEL regions:**
  - Just having multiple threads does not mean work is shared!
  - Only three work sharing constructs: DO (Fortran)/FOR(C), SECTION, and SINGLE

OpenMP Structure

- **Worksharing**
  - do/for, sections, single
- **Synchronization**
  - barrier, critical, ordered,master, atomic, flush
- **Data scoping**
  - shared, private, firstprivate, lastprivate, threadprivate, reduction
- **OpenMP library**
  - OMP_GET_NUM_THREADS(), etc
- **OpenMP environment variables**
  - OMP_NUM_THREADS, etc.


OpenMP
Directives
Binding
Nesting
Conditional Compilation
Control Constructs
Parallel Region
If()
Work Sharing
Do
Schedule()
Ordered
Sections
Single
Data Constructs
ThreadPrivate
Data Scope
Shared()
Private()
FirstPrivate()
LastPrivate()
Reduction()
Copyin()
Default()
Synchronization Constructs
Master
Critical
Barrier
Atomic
Ordered
Runtime Library
Environment Functions
Lock Functions
Environment Variables
OMP_SCHEDULE
Static
Dynamic,chunk
Guided,chunk
OMP_NUM_THREADS
OMP_DYNAMIC
OMP_NESTED

Parallelizing Procedure

1. Profile to locate candidate parallel loop(s)
2. Analyze variables and code
3. Insert directives and/or restructure
4. Compile and run program
5. Debug

Work Sharing

- **DO/FOR**
  - $OMP DO in Fotran, but can be combined with PARALLEL: $OMP PARALLEL DO (and closed with $OMP END PARALLEL)
  - #pragma omp for in C
- **SECTION**
  - ```
    #pragma omp parallel sections
    {
        #pragma omp section
        phase1();
        #pragma omp section
        phase2();
        #pragma omp section
        phase3();
    }
    ```
    ```