OpenMP

Shweta Deshmukh

Center for Development of Advanced Computing

Pune

shwetad@cdac.in

OpenMP follows …

Explicit parallelism

Shared Memory model

Developing a parallel Application

All is done only to get speedup

Think and have clear idea about program

for(i=0;i<=?;i++)
{ }

?

for( )
{ }

OpenMP stands here → Implementation of algorithm

Topics to be covered…

- Why OpenMP
- History
- What is OpenMP
- Key Features
- OpenMp constructs
- Pros and Cons

Why should I use OpenMP ??

Why OpenMP?

- Portable
- Implemented incrementally
- It is not intrusive
- supports data parallelism

- ✓ Why OpenMP
- o History
- o What is OpenMP
- o Key Features
- o OpenMp constructs
- o Pros and Cons

History

- The OpenMP Architecture Review Board (ARB) published its first standard, OpenMP for FORTRAN 1.0, in October of 1997.
- 1998 : version 1.0 of C/C++
- 2000 : version 2.0 of Fortran
- 2002 : version 2.0 of C/C++
- 2005 : version 2.5 of C/C++/Fortran

- ✓ Why OpenMP
- ✓ History
- o What is OpenMP
- o Key Features
- o OpenMp constructs
- o Pros and Cons

OpenMP !!!…What is it?

- Specification for a set of compiler directives, library routines and environment variables that can be used to specify shared memory parallelism.
- It may be used to explicitly direct multi-threaded, shared memory parallelism
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

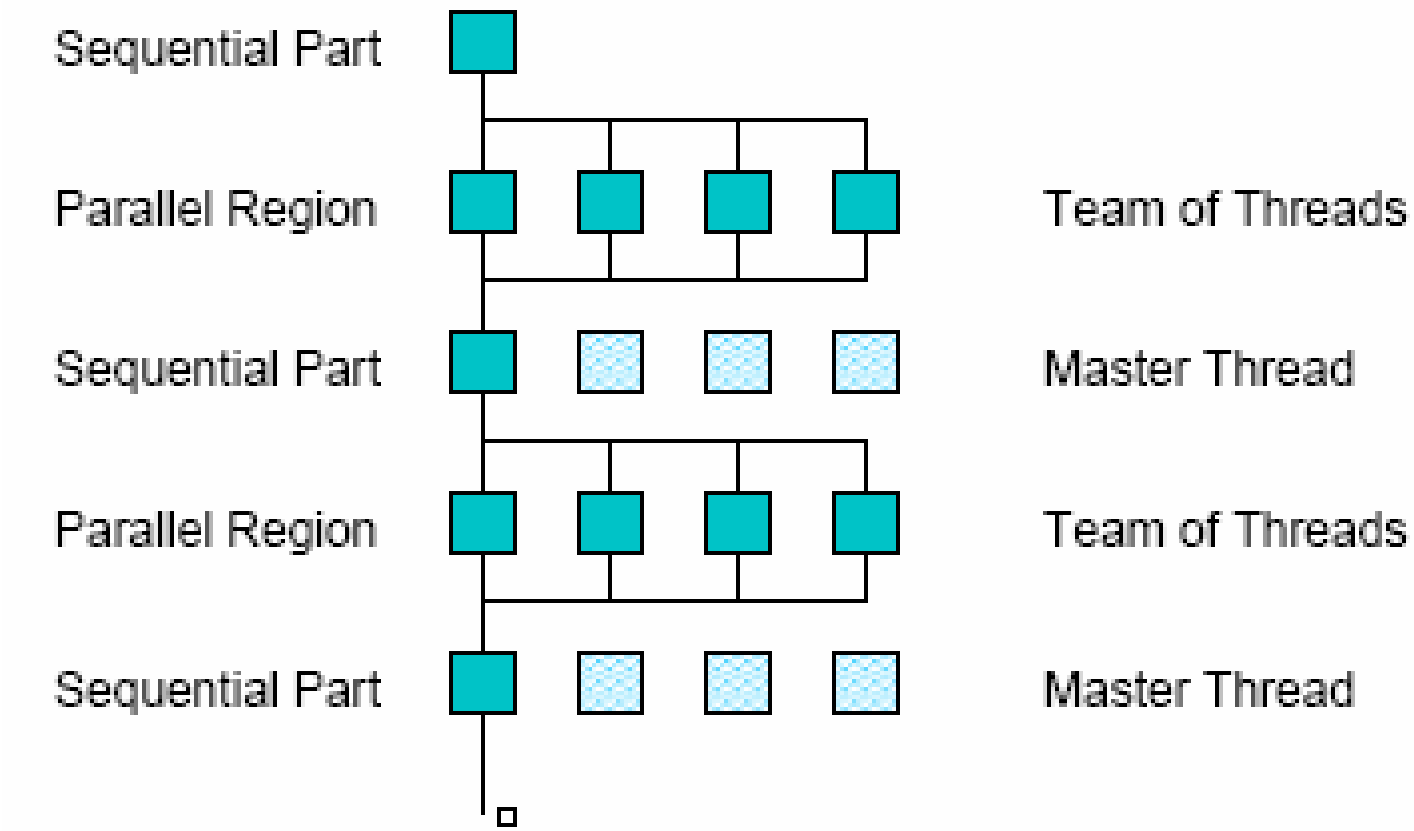
- Workload is distributed between threads

  Variables can be
  - **shared among all threads**
  - **duplicated for each thread**

  Threads communicate by sharing variables
  - Unintended sharing of data can lead to race conditions
- To control race conditions:

  Use synchronization to protect data conflicts.
- Careless use of synchronization can lead to dead-locks

Master-worker team thread pattern



Simple OpenMP program

- Most OpenMP constructs are compiler directives or pragmas
- The focus of OpenMP is to parallelize loops
- OpenMP offers an incremental approach to parallelism

Simple Program

```
#include<omp.h>

int main()
{
        int Res[1000];

    #pragma omp parallel for

        for(int i=0;i<1000;i++)
        {

        do_huge_comp(Res[i]);
        }
        return 0;
}
```

```
      PROGRAM ABC
      INTEGER Res(1000)

!$OMP PARALLEL DO
      DO 10 i=0,1000
            do_huge_comp(Res(i))
10    CONTINUE
!$OMP END PARALLEL DO
      END PROGRAM
```

- ✓ Why OpenMP
- ✓ History
- ✓ What is OpenMP
  - o Key Features
  - o OpenMp constructs
  - o Pros and Cons

Key Features

- OpenMP delivers single source portability for shared-memory parallelism
- User parallelize the main program using one or more parallel directives,and use other directives to control execution in the parallel subroutines.
- It provides a set of run-time library routines with associated environment variables

Key Features

- OpenMP provides more flexible functionality to control data environment
- The clause DEFAULT(PRIVATE) directs all variables in a parallel region to be private
- It also introduces an ATOMIC directive which allows compiler to take advantage of most efficient scheme to implement atomic updates to a variable

- ✓ Why OpenMP
- ✓ History
- ✓ What is OpenMP
- ✓ Key Features
  - o OpenMp constructs
  - o Pros and Cons

Compiler directives:

C/C++:

```
#pragma omp directives[clause,…]
```

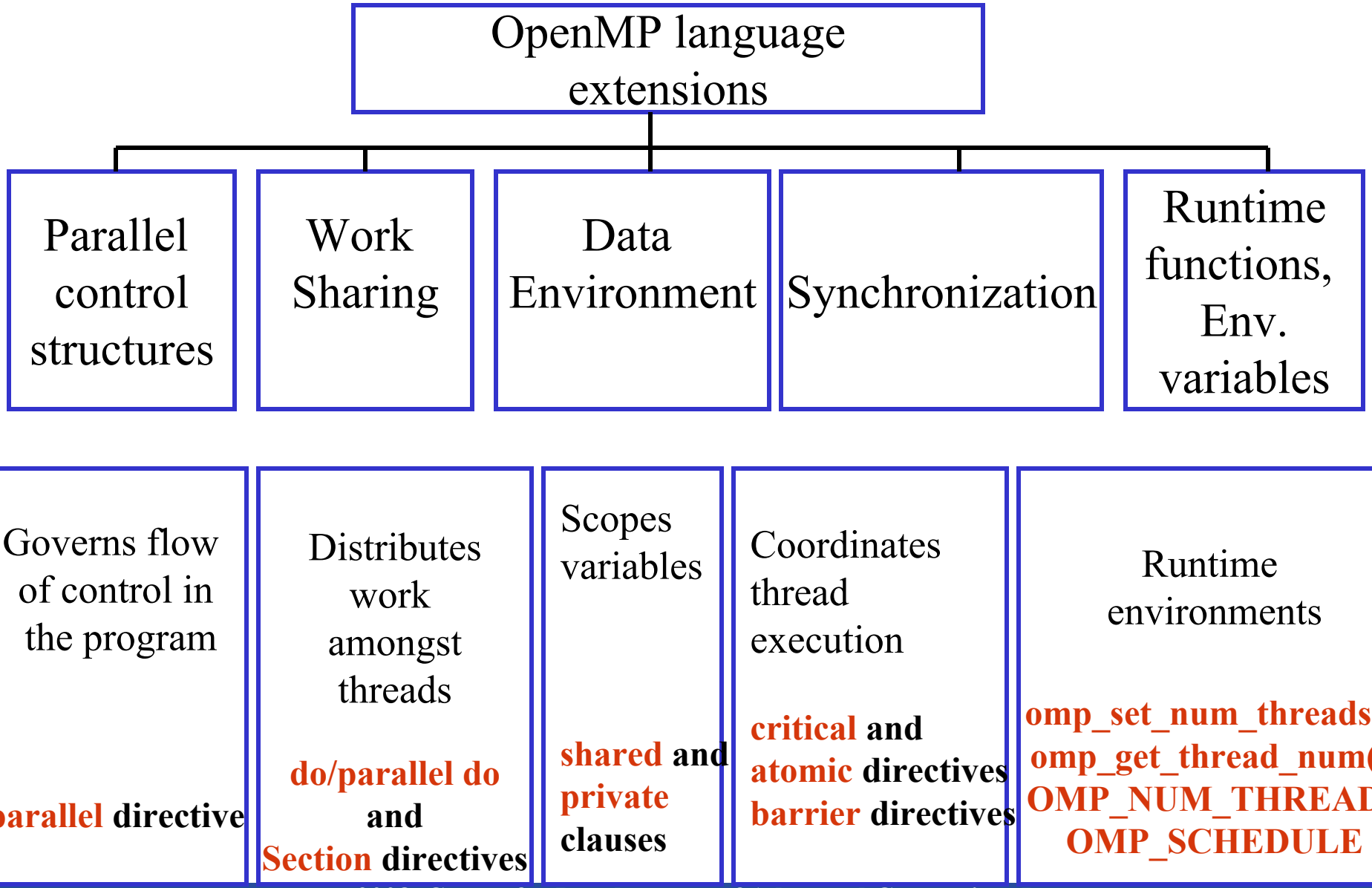
The include file:

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

Fortran:

```
!$OMP directive[clause,…]
C$OMP directive[clause,…]
*$OMP directive[clause,…]
```

OpenMP constructs



C/C++ general code structure

```
# include <omp.h>
main() {
int var1, var2, var3;
serial code
……
Beginning of parallel region, fork a team of threads.
Specify variable scoping
#pragma omp parallel private (var1, var2), shared(var3)
{
Parallel region executed by all threads
……..
……..
All threads join master thread and disband
}
Resume Serial code
}
```

Fortran General Code Structure

```
PROGRAM HELLO
INTEGER VAR1, VAR2, VAR3
Serial code
.....
.....
Beginning of parallel region, fork a team of threads.
Specify variable scoping
!$OMP PARALLEL PRIVATE (VAR1, VAR2),SHARED(VAR3)
Parallel region executed by all threads
.....
.....
All threads join master thread and disband
!$OMP END PARALLEL
.....
Resume Serial code
END
```

OpenMP parallel region construct

Block of code to be executed by multiple threads in parallel.

Each thread executes the **same code redundantly**!

C/C++:

```
#pragma omp parallel [ clause [ clause ] ... ]
 {    structured block    }
```

Fortran:

```
!$OMP PARALLEL[clause]
!$OMP END PARALLEL
```

clause can be one of the following:

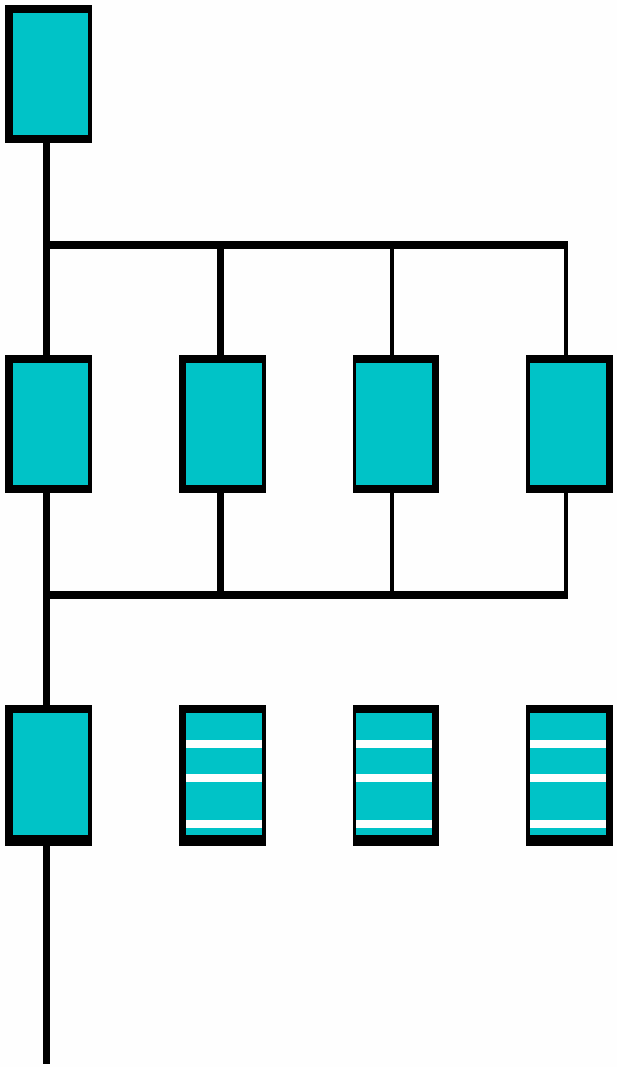
private(list)

shared(list)

OpenMP constructs

```
#pragma omp parallel
int main(int argc, char* argv[])
{
   #pragma omp parallel
     printf("Hello,world.\n");
    return 0;
}
```

```
        PROGRAM HELLO_WORLD
!$OMP PARALLEL
        print *,'hello world'
!$OMP END PARALLEL
        END HELLO_WORLD
```



OpenMP constructs

Work sharing

used to specify how to assign independent work to one or all of the threads.

i.e Organization of parallel work

- *omp for* or *omp do*: used to split up loop iterations among the threads
- *sections*: assigning consecutive but independent code blocks to different threads
- *single*: specifying a code block that is executed by only one thread, a barrier is implied in the end
- *master*: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

Detailed Syntax

For Loop

#pragma omp for[clause clause[]…]
for loop

where each clause is one of

- private(list)
- firstprivate(list)
- lastprivate(list)
- reduction(operator: list)
- ordered
- schedule(kind, [chunk_size])
- nowait

Do Loop

!$omp do[clause clause[]…]
do loop
[!$omp end do[nowait]]

where each clause is one of

- private(list)
- firstprivate(list)
- lastprivate(list)
- reduction(operator: list)
- ordered
- schedule(kind, [chunk_size])
- nowait

OpenMP for construct

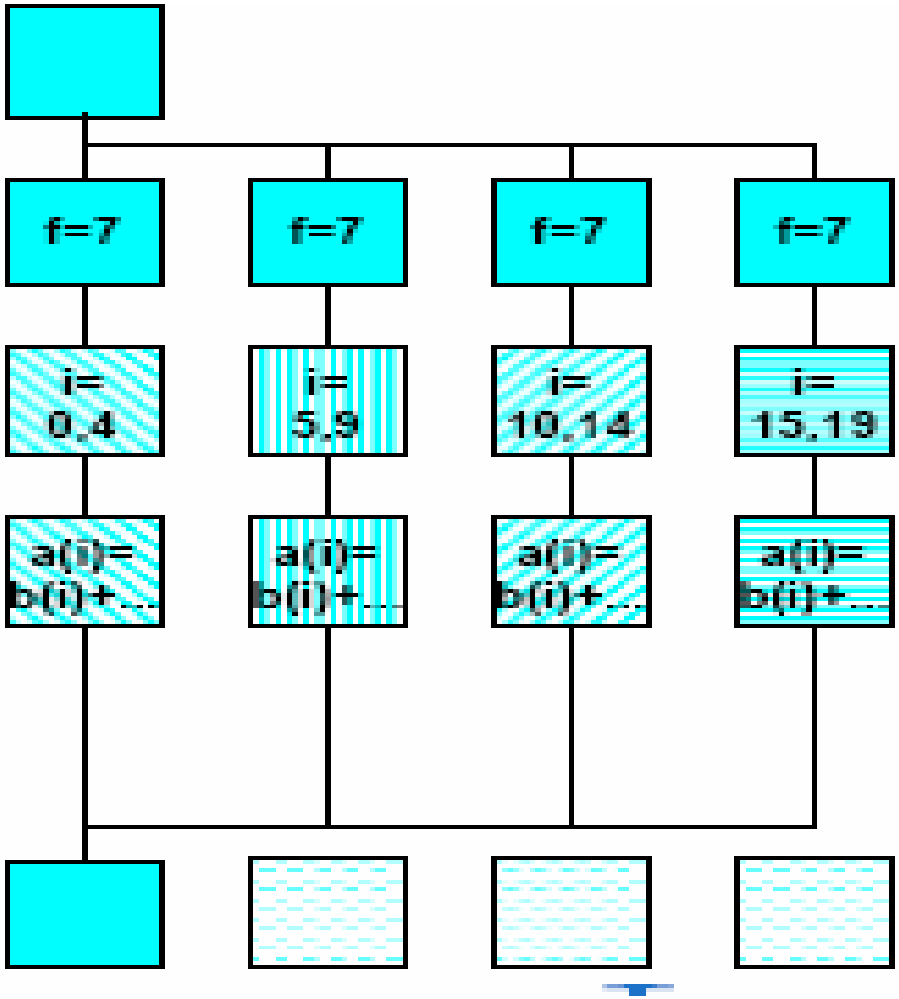
C Structure

```
#pragma omp parallel
private(f)
{
  f=7;
#pragma omp for
for (i=0; i<20; i++)
        a[i] = b[i] + f * (i+1);
} /* omp end parallel */
```

Fortran Structure

```
        PROGRAM
         INTEGER a(20),b(20),f,i
!$OMP PARALLEL PRIVATE(f)
        f=7
!$OMP DO
        do 100 i = 0 , 20
                a(i) = b(i) + f * (i+1)
100     CONTINUE
!$OMP END DO
!$OMP END PARALLEL
        END PROGRAM
```

OpenMP constructs



Implicit barrier at the end of the for loop, can be disabled with the **nowait** clause

Firstprivate

#pragma omp parallel firstprivate(list)

Firstprivate is the superset of the functionality provided by private clause.All the members of list are private(local to each thread) and initialized to the value in the preceding serial code.

First Private Example

Fortran structure

```
      PROGRAM FIRSTPRIVATE
      INTEGER myid,i,j
      INTEGER omp_get_thread_num
      i = 123 ; j = 455
      call omp_set_num_threads(2)
!$OMP PARALLEL DEFAULT(PRIVATE)
      myid=omp_get_thread_num()
      print*,'myid =',myid,'i = ',i,'j = ',j
!$OMP END PARALLEL
      print*,'.I am in serial region.'
!$OMP PARALLEL DEFAULT(PRIVATE) &
      FIRSTPRIVATE(i,j)
      myid=omp_get_thread_num()
      print*,'myid=',myid, 'i=',i, 'j=',j
!$OMP END PARALLEL
      END PROGRAM
```

Output

```
myid =          0 i =          0 j =          0
myid =          1 i =          0 j =          0
.I am in serial region.
myid=           0 i=         123 j=         455
myid=           1 i=         123 j=         455
```

Lastprivate

#pragma omp parallel lastprivate(list)

- Like private within the parallel construct - each thread has its own copy.
- The value corresponding to the last iteration of the loop(in serial mode) is saved following the parallel construct.

```
!$omp do shared(x)
!$omp lastprivate(I)
      do I =1 to N
            x(I)=0
      end do
      n=I
```

When the loop is finished,I is saved to the value corresponding to the last iteration in serial mode(i.e n=N+1)

OpenMP constructs

schedule clause specifies how iterations of the loop are divided among the threads of the team.

OpenMP supports four scheduling classes:

- static
- dynamic
- guided
- runtime

OpenMP constructs

schedule(static [,chunk])

Threads get a chunk of data to iterate over

schedule(dynamic [,chunk])

Threads grab chunk iterations off work queue until all work is exhausted

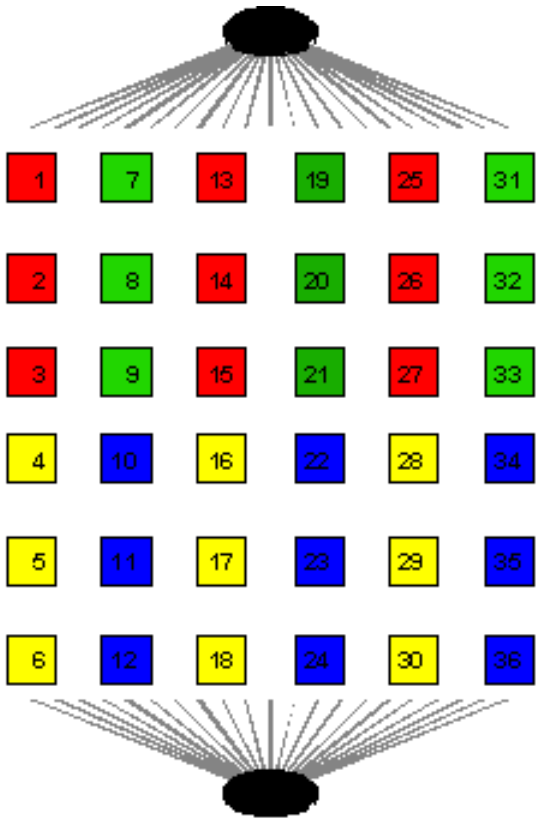
schedule(guided [,chunk])

Threads grab large chunk sizes and decreases to specified chunk size as the computation progresses

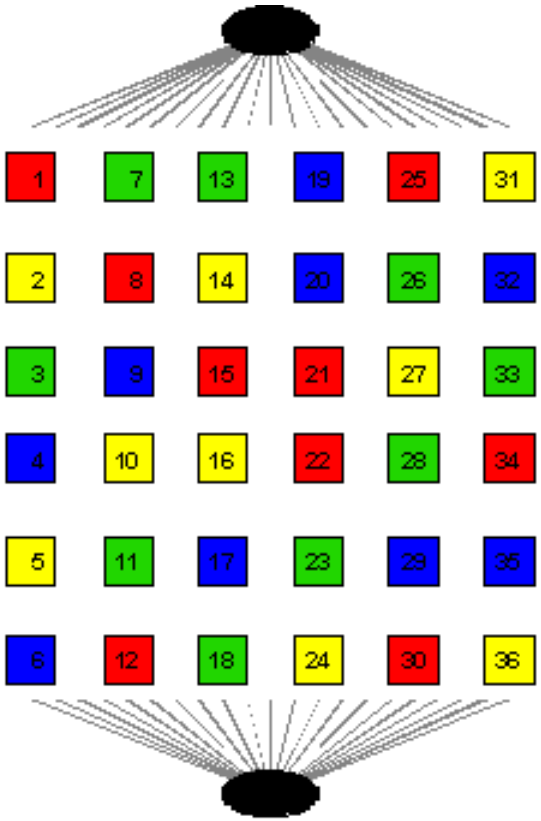
schedule(runtime)

Use the schedule defined at runtime by the OMP_SCHEDULE environment variable

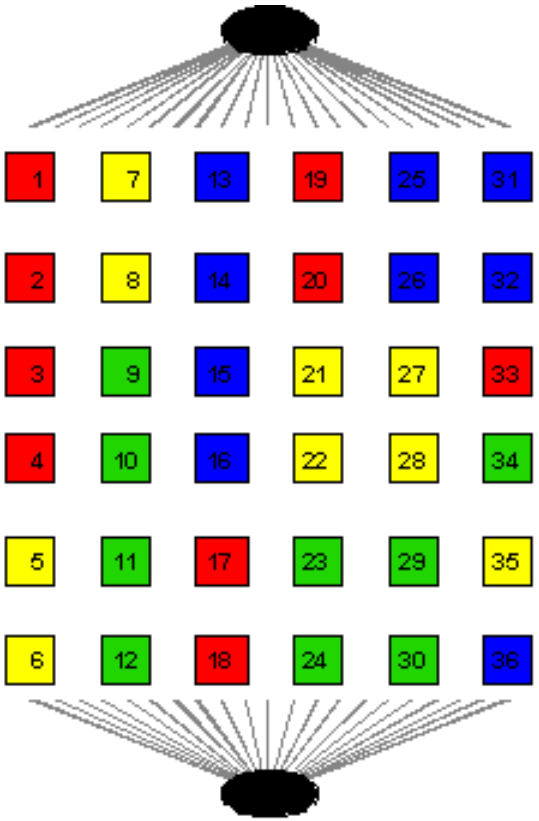
Loop Scheduling



Static (3)



Dynamic(1)

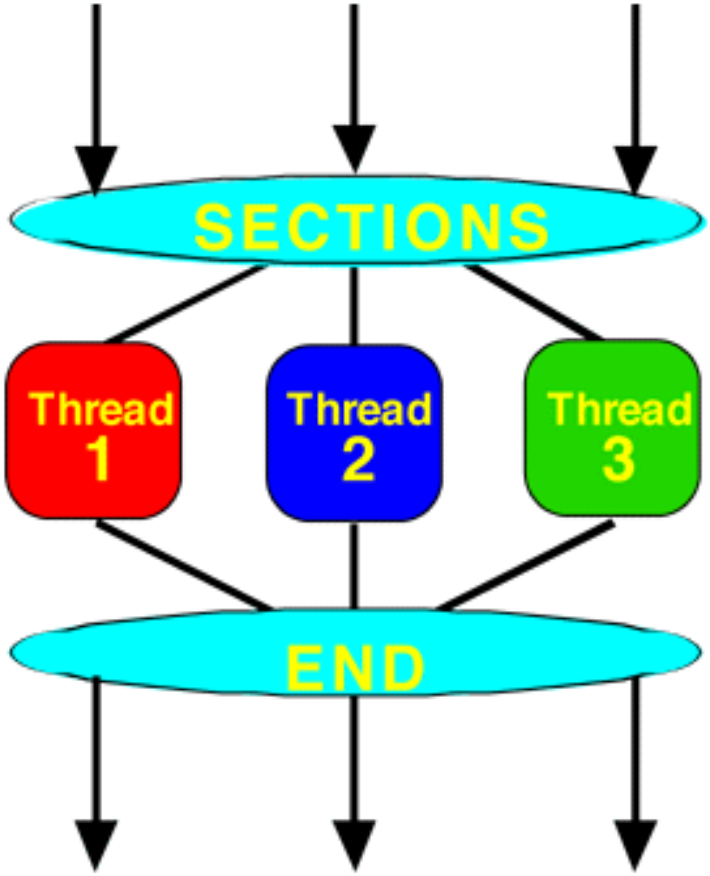


Guided(1)

$$\pi_k = \left\lceil \frac{\beta_k}{2N} \right\rceil$$

Section directive

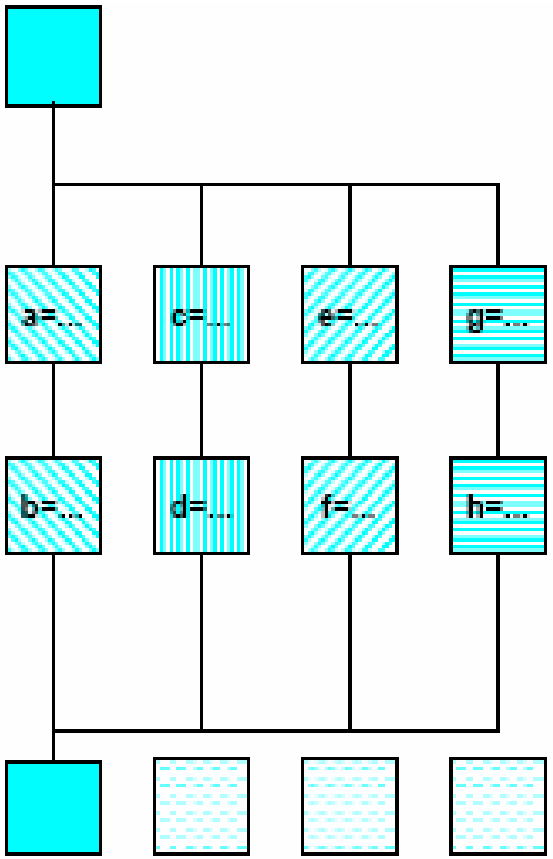
```
#pragma omp parallel
   #pragma omp sections nowait
   {
        #pragma omp section
                thread1_work();
        #pragma omp section
                thread2_work();
         #pragma omp section
                thread3_work();
   }
```



OpenMP Section Directive

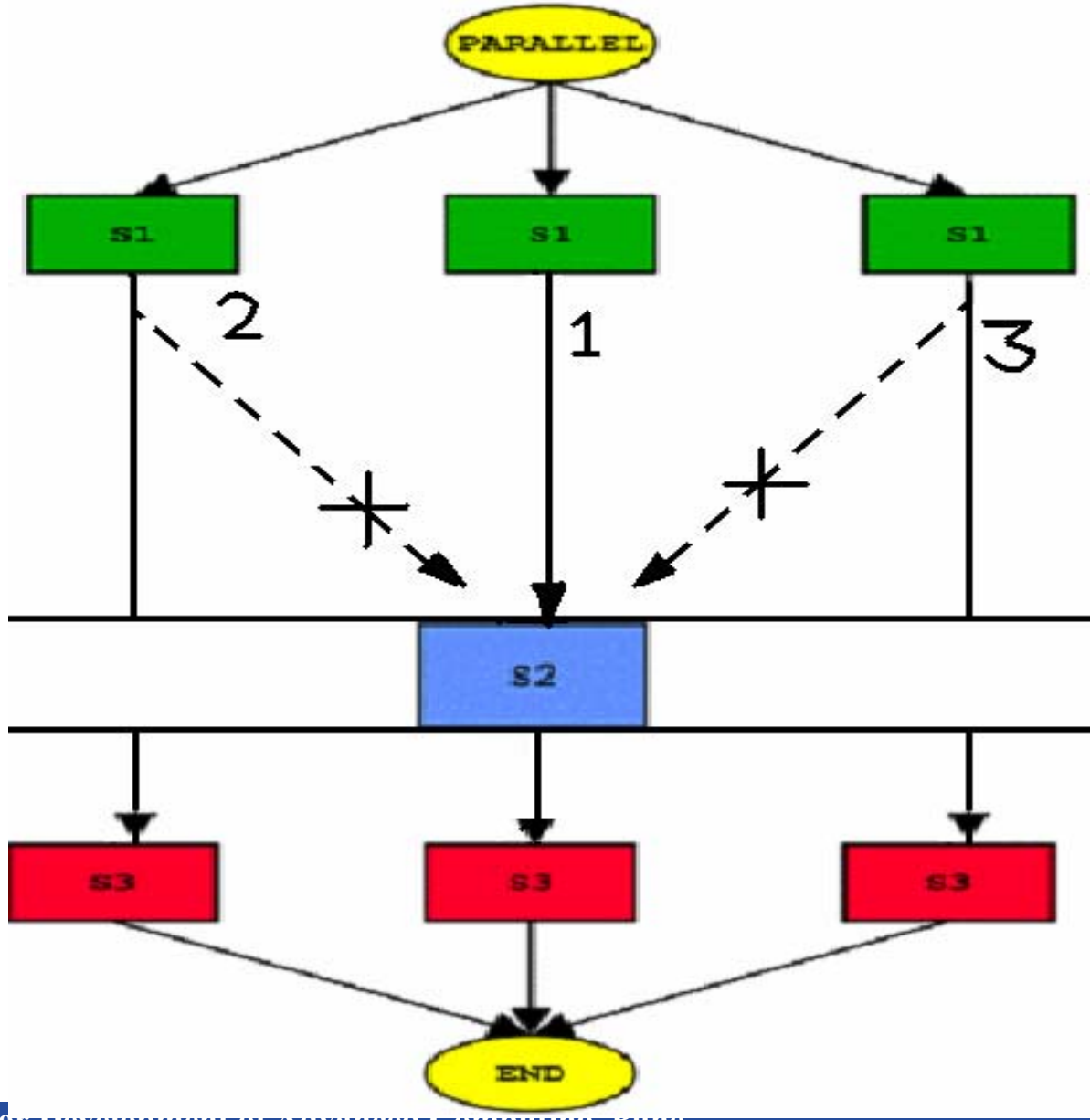
```
#pragma omp parallel
{
#pragma omp sections
     {{ a=...;
     b=...; }
#pragma omp section
     { c=...;
     d=...; }
#pragma omp section
     { e=...;
     f=...; }
#pragma omp section
     { g=...;
     h=...; }
} /*omp end sections*/
} /*omp end parallel*/
```

```
!$OMP PARALLEL
!$OMP SECTIONS
          a=…;
          b=…;
!$OMP SECTION
          c=…;
          d=…;
!$OMP END SECTION
 !$OMP SECTION
          e=…;
          f=…;
!$OMP END SECTION
!$OMP SECTION
          g=...;
          h=...;
 !$OMP END SECTION
 !$OMP END SECTIONS
```



Single directive

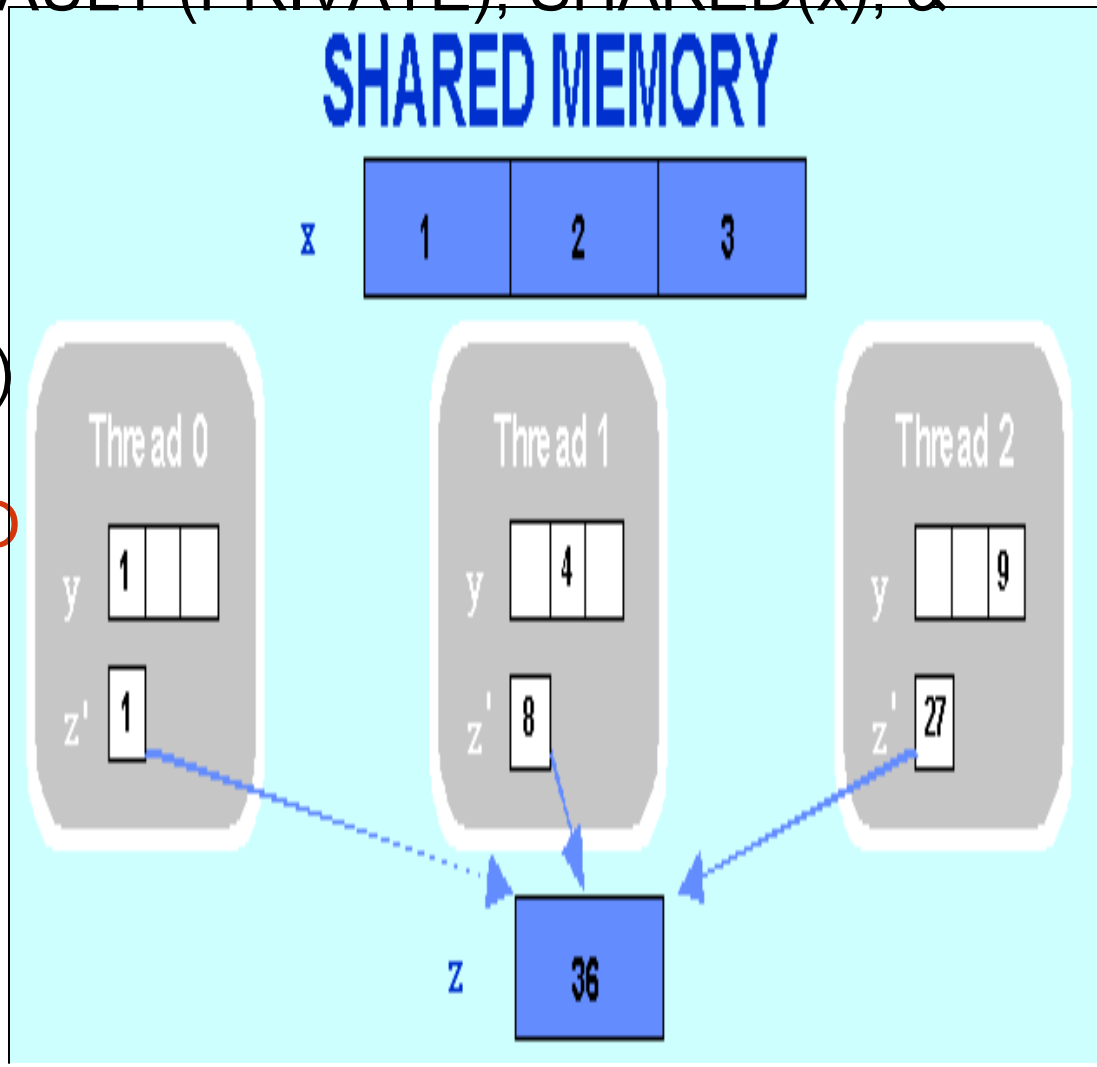
```
!$OMP PARALLEL
      CALL S1
      !$OMP SINGLE
            CALL S2
      !$OMP END SINGLE
      CALL S3
!$OMP END PARALLEL
```



Data environment

- *shared*: the data within a parallel region is shared, which means visible and accessible by all threads simultaneously.
- *private*: This means each thread will have a local copy and use it as a temporary variable.
- *default*: allows the programmer to state that the default data scoping within a parallel region will be either *shared*, *private*, or *none*. The *none* option forces the programmer to declare each variable in the parallel region as either shared or private.

```
INTEGER x(3), y(3), z
!$OMP PARALLEL DO DEFAULT (PRIVATE), SHARED(x), &
 !$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
```



OpenMP constructs

Synchronization

- *critical section*
- *atomic*
- *ordered*
- *barrier*
- *nowait*

Critical section

Enclosed code executed by all threads, but **restricted to only one thread at a time**

C/C++:

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

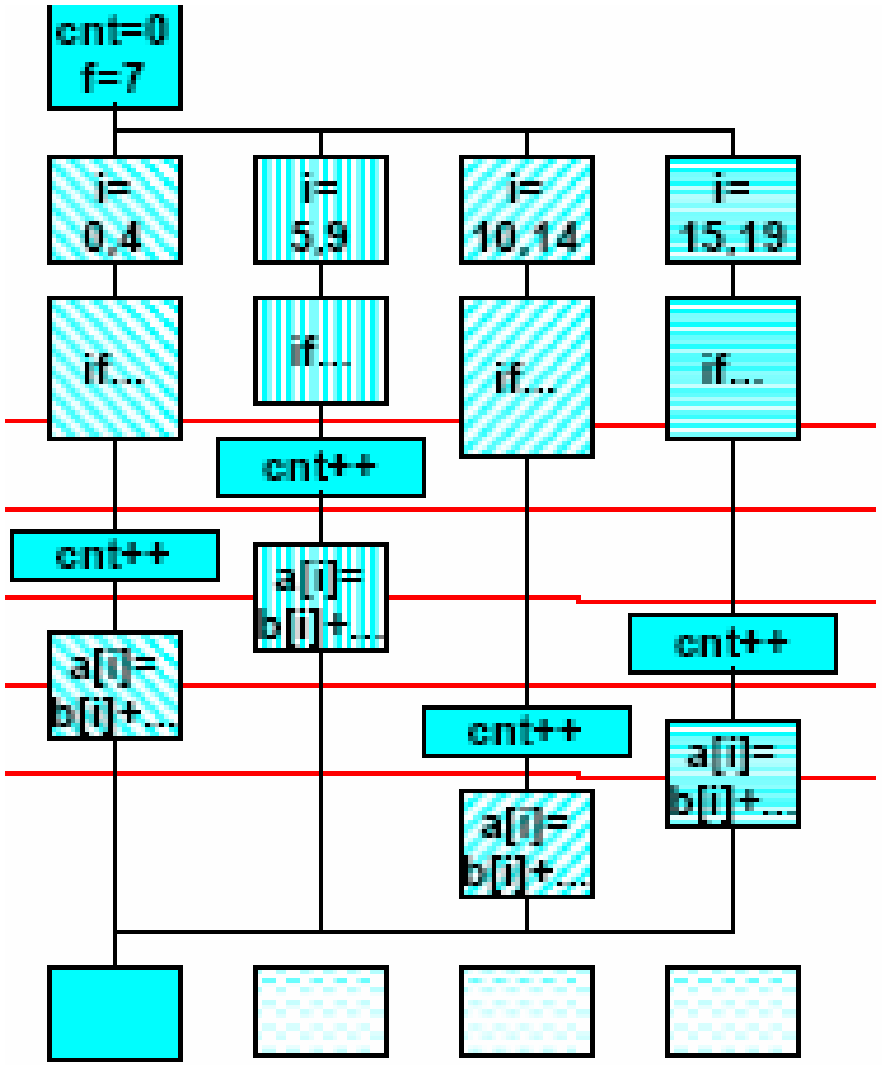
Fortran:

```
!$OMP CRITICAL
    structured block
!$OMP END CRITICAL
```

A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name.

OpenMP critical

```
cnt = 0;
f=7;
#pragma omp parallel
{
#pragma omp for
     for (i=0;i<20;i++)
     {
          if (b[i] == 0)        {
   #pragma omp critical
                    cnt ++;
          } /* endif */
     a[i] = b[i]+f*(i+1);
     } /* end for */
} /*omp end parallel */
```



OpenMP critical

Fortran code snippet

```
        cnt=0
        f=7
!$OMP PARALLEL
!$OMP DO
        do 20 i=0,20
                if(b(i) .eq. 0) then
!$OMP CRITICAL
        cnt = cnt + 1
!$OMP END CRITICAL
        a(i) = b(i) + f * (i + 1)
!$OMP END DO
!$OMP END PARALLEL
```

OpenMP Ordered

```
#include <stdio.h>
#include <omp.h>
static float a[1000], b[1000], c[1000];
void test2(int iter)
{
     printf("test2() iteration %d\n", iter);
}
int main( )
{
    int i;
     #pragma omp parallel
     {
            #pragma omp for ordered
                    for (i = 0 ; i < 5 ; i++)
                            test2(i);
 } }
```

output

test2() iteration 0
test2() iteration 1
test2() iteration 2
test2() iteration 3
test2() iteration 4

Barrier and nowait

```
#pragma omp parallel shared (A, B, C)
private(d)
{
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_calc3(id);
}
```

Environment Variables

OMP_NUM_THREADS

- sets the number of threads to use during execution
- when dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use

OMP_NESTED

- Allow nesting of parallel region

Environment variables cont…

OMP_SCHEDULE

- applies only to do/for and parallel do/for directives that have the schedule type RUNTIME
- sets schedule type and chunk size for all such loops

OMP_DYNAMIC

- Allow run time system to determine the number of threads

Compilation of OpenMP program

Using Intel Fortran or C compiler

ifort <program name> -openmp –o <name of executable>

icc <program name> -openmp –o <name of executable>

Using GNU Fortran or C compiler

gfortran<program name> -fopenmp –o <name of executable>

gcc<program name> -fopenmp –o <name of executable>

Executing the OpenMP Program

Running a OpenMP Program

./ name of the executable

For example: running a hello world program

./ hello_world.out

- ✓ Why OpenMP
- ✓ History
- ✓ What is OpenMP
- ✓ Key Features
- ✓ OpenMp constructs
- o Pros and Cons

Pros:

- Simple
- Data layout and decomposition is handled automatically by directives.
- Unified code for both serial and parallel applications
- Original (serial) code statements need not, in general, be modified when parallelized with OpenMP.

***Cons*:**

- Currently only runs efficiently in shared-memory multiprocessor platforms
- Requires a compiler that supports OpenMP.
- Scalability is limited by memory architecture.
- Reliable error handling is missing.

References

- Introduction to Parallel Computing by Ananth Grama
- https://computing.llnl.gov/tutorials/openMP
- http://en.wikipedia.org/wiki/OpenMP
- Programming in OpenMP by Rohit Chandra

Thank you

Thank you

Thank you

Thank you

Thank you

Thank you

Thank you

Thank you

Thank You

Thank you