Introduction to OpenMP

Kadin Tseng
Scientific Computing and Visualization Group
Boston University
What is OpenMP?

- OpenMP is primarily a directive-based method to invoke parallel computations on shared-memory multiprocessor UNIX-, Linux- and Windows-based computers.
What is OpenMP?

- OpenMP is primarily a directive-based method to invoke parallel computations on many shared-memory multiprocessor UNIX-, Linux-, and Windows-based computers.
- OpenMP is available for fortran 77/90 and C/C++.
Why OpenMP?

• OpenMP is portable -- supported by Compaq, HP, IBM, Intel, SGI, SUN and others on UNIX, Linux, and NT.
Why OpenMP?

- OpenMP is portable -- supported by Compaq, HP, IBM, Intel, SGI, SUN and others on UNIX and NT.
- OpenMP can be implemented incrementally -- one subroutine (function) or even one do (for) loop at a time.
Why OpenMP?

- OpenMP is portable -- supported by Compaq, HP, IBM, Intel, SGI, SUN and others on UNIX and NT.
- OpenMP can be implemented incrementally -- one subroutine (function) or even one do (for) loop at a time.
- OpenMP is not intrusive (to original serial code) -- instructions appear in comment statements for fortran and through pragmas for C/C++.

Fortran 77:

```fortran
C$omp parallel do
  do i=1,n
    ...
  enddo
```

C/C++:

```c
#pragma parallel for
for (i=0; i<n; i++) {
  ...
}
```
OpenMP Methods

OpenMP

Library

Directives

Env Variables
OpenMP Methods

- OpenMP provides the programmer three complimentary methods to communicate with the compiler and runtime system on parallel processing:

```c
!$ me=omp_get_thread_num()
!$ call omp_set_num_threads(16)
```
OpenMP Methods

- OpenMP provides the programmer three complimentary methods to communicate with the compiler and runtime system on parallel processing:

  ![Diagram of OpenMP methods]

  ```
  !$omp parallel do
  do j = 1, n
      ...
  enddo
  ```
OpenMP Methods

- OpenMP provides the programmer three complimentary methods to communicate with the compiler and runtime system on parallel processing:

  - Library
  - Directives
  - Env Variables

```
setenv OMP_NUM_THREADS 16
```
OpenMP Methods

- OpenMP provides the programmer three complimentary methods to communicate with the compiler and runtime system on parallel processing:

```
!$omp call omp_set_num_threads(8)
```

```
setenv OMP_NUM_THREADS 16
```

The library call overrides value set by environment variable OMP_NUM_THREADS
Serial and Parallel Regions -- 1

Master Thread (0)

Thread 1
Thread 2
Thread 3

Parallel regions
Parallel regions where child threads:
> spawned upon entering
> released upon exiting

Serial regions executed by master thread

Master Thread 0
Active throughout duration of job
Master thread, performs tasks in both serial and parallel regions.
OpenMP Rules and Conventions

Some Rules of OpenMP directives for f77, f90 and C/C++

• clauses -- default, private, shared
• line continuation -- follows rule for language
• make OpenMP lines invisible for serial processing
  • C$ for F77
  • !$ for F90
  • use #ifdef for C/C++
OpenMP Rules and Conventions

Rules of OpenMP directives for f77, f90 and C/C++:

Fortran 77:
c$omp parallel default(none)
c$omp&private(i,j) shared(a)

construct: parallel
clauses: default, private, shared
Rules of OpenMP directives for f77, f90 and C/C++:

Fortran 77:
c$omp parallel default(none)
c$omp&private(i,j) shared(a)

Fortran 90:
!$omp parallel default(none) &
!$omp private(i,j) shared(a)

construct: parallel
clauses: default, private, shared
OpenMP Rules and Conventions

Rules of OpenMP directives for f77, f90 and C/C++:

Fortran 77:
c$omp parallel default(none) c$omp&private(i,j) shared(a)

Fortran 90:
!$omp parallel default(none) & !$omp private(l,j) shared(a,b)

C/C++:
#pragma omp parallel default(none) \ private(i,j) shared(a)

construct: parallel
clauses: default, private, shared

#include <stdio.h>
define n 10
#define m 20

void main()
{
ipint i, j, a[n][m];
#pragma omp parallel default(none) \
private(i,j) shared(a)
#pragma omp for
for (j=0; j<m; j++) {
    for (i=0; i<n; i++) {
        a[i][j] = i+(j-1)*n;
    }
}
}
Basics - parallel & do directives

OpenMP directives used:
• c$omp parallel [clauses]
• c$omp end parallel

program f77_parallel
  implicit none
  integer n, m, i, j
  parameter (n=10, m=20)
  integer a(n,m)

  c$omp parallel default(none)
  c$omp& private(i,j) shared(a)
  do j=1,m
    do i=1,n
      a(i,j) = i+(j-1)*n
    enddo
  enddo
  c$omp end parallel
end
OpenMP directives used:
- `c$omp parallel [clauses]`
- `c$omp end parallel`
  - parallel clauses include:
    - default(none|private|shared)
    - private(…)
    - shared(…)

Default setting

```f77
program f77_parallel
    implicit none
    integer n, m, i, j
    parameter (n=10, m=20)
    integer a(n,m)

    c$omp parallel default(none)
    c$omp& private(i,j) shared(a)
    do j=1,m
        do i=1,n
            a(i,j) = i+(j-1)*n
        enddo
    enddo
    c$omp end parallel
end
```

program f77_parallel
    implicit none
    integer n, m, i, j
    parameter (n=10, m=20)
    integer a(n,m)

    c$omp parallel default(none)
    c$omp& private(i,j) shared(a)
    do j=1,m
        do i=1,n
            a(i,j) = i+(j-1)*n
        enddo
    enddo
    c$omp end parallel
end
OpenMP directives used:
- `c$omp parallel [clauses]`
- `c$omp end parallel`
  - parallel clauses include:
    - `default(none|private|shared)`
    - `private(...)`
    - `shared(...)`

Program:

```f77
program f77_parallel
  implicit none
  integer n, m, i, j
  parameter (n=10, m=20)
  integer a(n,m)

  c$omp parallel default(none)
  c$omp&private(i,j) shared(a,b)
  do j=1,m
    do i=1,n
      a(i,j) = i+(j-1)*n
    enddo
  enddo
  c$omp end parallel
end```

- Each arrow denotes 1 thread
- All threads perform identical task
OpenMP directives used:
• `c$omp parallel [clauses]`
• `c$omp end parallel`
  – parallel clauses include:
    › default(none|private|shared)
    › private(…)
    › shared(…)
• `c$omp do`

With default scheduling:
Thread 0 works on j=1:5
Thread 1 works on j=6:10
Thread 2 works on j=11:15
Thread 3 works on j=16:20
Basics - parallel do directive

- `c$omp parallel do [clauses]`
- `c$omp end parallel do`
  - parallel do clauses include:
    - default(none|private|shared)
    - private(…)
    - shared(…)

```f77
program f77_Parallel_Do
  implicit none
  integer n, m, i, j
  parameter (n=10, m=20)
  integer a(n, m)

  c$omp parallel do private(i, j)
      do j=1, m
          do i=1, n
              a(i, j) = i + (j - 1) * n
          enddo
      enddo
  c$omp end parallel do
end
```
SECTIONS assigns one thread to each SECTION.

```fortran
PROGRAM SINGLE
  IMPLICIT NONE
  INTEGER :: omp_get_thread_num, myid
  !$OMP PARALLEL PRIVATE(myid)
  !$OMP SECTIONS
    !$OMP SECTION
      myid = omp_get_thread_num()
      print*,'Section 1 is thread', myid
    !$OMP SECTION
      myid = omp_get_thread_num()
      print*,'Section 2 is thread', myid
    !$OMP SECTION
      myid = omp_get_thread_num()
      print*,'Section 3 is thread', myid
  !$OMP END SECTIONS
  !$OMP END PARALLEL
END PROGRAM SINGLE
```

If there are more SECTIONs than threads, some threads will be assigned multiple SECTIONs.
SECTIONS assigns one thread to each SECTION.

```c
#include <stdio.h>
void main() {
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      printf("Section 1 is thread %d\n",omp_get_thread_num());
      #pragma omp section
      printf("Section 2 is thread %d\n",omp_get_thread_num());
      #pragma omp section
      printf("Section 3 is thread %d\n",omp_get_thread_num());
    } } }
```

If there are more SECTIONs than threads, some threads will be assigned multiple SECTIONs.
• Code block enclosed by SINGLE directive is executed by a single thread. Remaining threads synchronize at end of SINGLE

```fortran
PROGRAM SINGLE
  IMPLICIT NONE
  INTEGER :: omp_get_thread_num, a, b, c, s
  !$OMP PARALLEL
  !$OMP SECTIONS
  !$OMP SECTION
    print*, 'Section 1 is thread', omp_get_thread_num(); a = 1
  !$OMP SECTION
    print*, 'Section 2 is thread', omp_get_thread_num(); b = 2
  !$OMP SECTION
    print*, 'Section 3 is thread', omp_get_thread_num(); c = 3
  !$OMP END SECTIONS
  !$OMP SINGLE
    print *, 'SINGLE thread is', omp_get_thread_num(), '; s = ', s
  !$OMP END SINGLE
  !$OMP END PARALLEL
END PROGRAM SINGLE
```
• Unlike SINGLE, only the master thread execute code block
• No barrier or synchronization

```
PROGRAM MASTER
   IMPLICIT NONE
   INTEGER :: omp_get_thread_num

   !$OMP PARALLEL
   !$OMP SECTIONS
   !$OMP SECTION
       print*, 'Section 1 is thread', omp_get_thread_num()
   !$OMP SECTION
       print*, 'Section 2 is thread', omp_get_thread_num()
   !$OMP SECTION
       print*, 'Section 3 is thread', omp_get_thread_num()
   !$OMP END SECTIONS
   !$OMP MASTER
       print *, 'Thread used in MASTER is', omp_get_thread_num()
   !$OMP END MASTER
   !$OMP END PARALLEL
END PROGRAM MASTER
```
Critical – Fortran 90

- Critical section is executed on every thread – one at a time

```fortran
program critical
  implicit none
  integer n
  n = 0

  !$omp parallel default(shared)
  !$omp critical
    call count(n)
  !$omp end critical
  !$omp end parallel
  print*, 'Number of processors =', n
end program critical

subroutine count(n)
  implicit none
  integer :: n
  n = n + 1
  return
end
```
**Critical - C**

- Enclosed section is executed on every thread – one at a time

```c
#include <stdio.h>
void main()
{
    int n = 0;
    #pragma omp parallel
    {
        #pragma omp critical
        count(&n); /* n is a shared variable */
    }
    printf("Number of processors = %d\n", n);
}
int count(int *n)
{
    *n += 1;
    return 0;
}
```
Ordered

- For use in conjunction with DO or PARALLEL DO
- Executes block of code in DO loop index order
- In addition to ORDERED directive within loop, ORDERED clause must also present in DO directive
- Ordered code block must not be branched into or out of

```fortran
program ordered
  implicit none
  integer i, x(10)

  !$omp parallel do default(shared) private(i) ordered
  do i = 1, 10
    x(i) = i
  !$omp ordered
  print *, 'x(', i, ') = ', x(i)
  !$omp end ordered
  end do
  !$omp end parallel do
end program ordered
```
Threadprivate/Copyin – F90

Program Threadprivate

```fortran
PROGRAM THREADPRIVATE
  IMPLICIT NONE
  INTEGER :: a, b, s1, s2
  COMMON/BLK1/a, b
!$OMP THREADPRIVATE(/BLK1/)  
a = 100; b = 200;
!$OMP PARALLEL COPYIN(a, b)
!$OMP SECTIONS
!$OMP SECTION
  s1 = a + b
  print*, 'a, b, s1=', a, b, s1
!$OMP SECTION
  s2 = a + b
  print*, 'a, b, s2=', a, b, s2
!$OMP END SECTIONS
!$OMP END PARALLEL
END PROGRAM THREADPRIVATE
```

- Applied to common blocks
- THREADPRIVATE gives each thread its own private copy of specified common block(s)
- COPYIN initializes each thread’s copy with the master thread’s values

With copyin:

Section 1 is thread 0
a, b, s1 =  100 200 300
Section 2 is thread 1
a, b, s2 =  100 200 300

Without copyin:

Section 1 is thread 0
a, b, s1 =  100 200 300
Section 2 is thread 1
a, b, s2 =  0 0 0
**Threadprivate – C**

Used to make global file scope variables (C/C++) local and persistent to a thread

```c
#include <stdio.h>
int a, b, s1, s2;
#pragma omp threadprivate(a, b)
int main()
{
#pragma omp parallel
{
#pragma omp master
{ a = 100; b = 200; }
}
#pragma omp parallel copyin(a, b)
{
#pragma omp sections
{
#pragma omp section
{ s1 = a + b;
 printf("a, b, s1=%d %d \n", a, b, s1);
}
#pragma omp section
{ s2 = a + b;
 printf("a, b, s2=%d %d \n", a, b, s2);
}
} }
return 0;
}
```
Directives - Atomic

!$omp atomic

When used, this directive prevents specific memory location from accessed by multiple threads concurrently in a parallel region. This directive affects only the statement immediately following it. Furthermore, only load and store in the statement will be affected by atomic.

```c
!$omp parallel do
    do i = 1, n
      !$omp atomic
      a(map(i)) = a(map(i)) + 1
      c(i) = d(i)  ! No effect
    enddo
!$omp end parallel do
```
Directives - Barrier

!$omp barrier

This directive synchronizes all threads in the parallel region. All threads in the region wait until every thread in the team have reached this location.
**Clauses - Firstprivate**

**Firstprivate(list)** is a superset of the functionality provided by the **private** clause. In addition to rules and semantics governing **private**, all members of **list** are initialized to values existed prior to construct.

```fortran
program firstprivate
  implicit none
  integer myid, i, j
  i = 123; j = 456
  !$omp parallel default(private)
  myid = omp_get_thread_num()
  print*, 'myid, i, j:', myid, i, j
  !$omp end parallel
  i = 321; j = 654
  !$omp parallel default(private) &
  firstprivate(i, j)
  myid = omp_get_thread_num()
  print*, 'myid, i, j:', myid, i, j
  !$omp end parallel
end program firstprivate
```

**Output**

<table>
<thead>
<tr>
<th>myid, i, j:</th>
<th>0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>myid, i, j:</td>
<td>1 0 0</td>
</tr>
<tr>
<td>myid, i, j:</td>
<td>0 321 654</td>
</tr>
<tr>
<td>myid, i, j:</td>
<td>1 321 654</td>
</tr>
</tbody>
</table>
Lastprivate Clause

Lastprivate(list) is a superset of the functionality provided by the private clause. Members of list are subject to the rules of private clause. When lastprivate is used in conjunction with do loop, the loop index contains the value of loop count plus 1 beyond the parallel region.

```
program lastprivate_example
  implicit none
  integer :: i, a(6), n=6, &
    myid, omp_get_thread_num
  !$omp parallel private(myid)
  myid = omp_get_thread_num()
  !$omp do lastprivate(i)
    do i=1,n
      a(i) = i
      print*, 'Without', &
      'lastprivate, myid, i= ', myid, i
    enddo
    print*, 'With', &
    'lastprivate, myid, i= ', myid, i
    print*, 'With lastprivate, i= ', i
  !$omp end parallel
print*,'With lastprivate, i= ', i
end program lastprivate_example
```

OUTPUT

Without lastprivate,myid,i= 0 1
Without lastprivate,myid,i= 1 1
Without lastprivate,i= 1
With lastprivate,myid,i= 1 7
With lastprivate,myid,i= 0 7
With lastprivate,i= 7
Reduction

- Reduction operations +, -, .AND., .OR., .EQV., .NEQV., MAX, MIN, IAND, IOR, and IEOR can be parallelized if reduction clause is used

```c
!$omp parallel do reduction(+:x,y) private(i)
do i = 1, n
   x = x + A(i)
   y = y + B(i)
enddo
!$omp end parallel do
```
• Operating system may reduce number of threads in a parallel region
• OMP_SET_DYNAMIC turns thread reduction feature on or off
  – overrides OMP_DYNAMIC environment variable

```fortran
!... number of threads in next loop must not be
!... reduced by the operating system
  if( omp_get_dynamic() ) then
    call omp_set_dynamic(.false.)
  endif

!$omp parallel do
  do i = 1, n
    myid = omp_get_thread_num()
    call sub1(myid,result(myid))
  enddo

!... turn dynamic thread reduction back on
  call omp_set_dynamic(.true.)
```
Nested parallelism ***may*** be available on a given system

**OMP_SET_NESTED** turns nested parallelism feature on or off

- overrides **OMP_NESTED** environment variable

```fortran
call omp_set_nested(.true.)
if( omp_get_nested() ) then
  !$omp parallel do
  do i = 1, n
    myid = omp_get_thread_num()
    call sub_with_nested_parallelism(myid)
  enddo
else
  !$omp parallel do
  do i = 1, n
    myid = omp_get_thread_num()
    call sub_without_nested_parallelism(myid)
  enddo
endif
```
integer function omp_get_max_threads() returns number of threads requested in serial or parallel region.

```fortran
program omp_get_max_threads_example
  implicit none
  ! Demonstrates usage of omp_get_max_threads()
  integer, parameter :: Nprocs=4
  call omp_set_num_threads(Nprocs)  !: set # of procs
  print*, ' In serial region, max_threads returns ', &
  omp_get_max_threads()
  !$omp parallel
  !$omp single
  print*, '********** parallel region **********'
  print*, ' In parallel region, max_threads returns ', &
  omp_get_max_threads()
  !$omp end single
  !$omp end parallel
end program omp_get_max_threads_example
```
integer function omp_get_num_threads() returns  
• requested number of threads in a parallel region  
• returns 1 in a serial region or a nested parallel region that is serialized.

```
program omp_get_num_threads_example
  implicit none
! This F90 program demonstrates the usage of omp_get_num_threads()
  integer, parameter :: Nprocs=4

  call omp_set_num_threads(Nprocs)  !! set # of procs
  print*, ' In serial region, omp_get_num_threads returns', &
    omp_get_num_threads()
!$omp parallel
!$omp single
  print*, ' In parallel region, omp_get_num_threads returns', &
    omp_get_num_threads()
!$omp end single
!$omp end parallel

end program omp_get_num_threads_example
```
integer function omp_get_num_procs() returns the number of physical processors available.

```
program omp_get_num_procs_example
  implicit none
  ! This F90 program demonstrates the usage of omp_get_num_procs()
  integer, parameter :: Nprocs=4

  call omp_set_num_threads(Nprocs)  !! set # of procs
  print*, ' In serial region, omp_get_num_procs returns', &
    omp_get_num_procs()

  !$omp parallel
  !$omp single
  print*, ' In parallel region, omp_get_num_procs returns', &
    omp_get_num_procs()
  !$omp end single
  !$omp end parallel

end program omp_get_num_procs_example
```
integer function omp_in_parallel() returns
• “true” -- inside a parallel region
• “false” -- otherwise.

program omp_in_parallel_example
implicit none
!
! Demonstrates usage of omp_in_parallel
integer, parameter :: Nprocs=4
!
call omp_set_num_threads(Nprocs)  !! set # of procs
print*, 'Am I in a parallel region (should be F)?', &
omp_in_parallel()
!
!$omp parallel
!$omp single

print*, '********** parallel region **********'
print*, ' Am I in a parallel region (T)? ', omp_in.parallel()
!
!$omp end single
!$omp end parallel
end program omp_in_parallel_example
setenv OMP_DYNAMIC $logical-value$

Here, $logical-value$ is either $true$ or $false$. The default setting is implementation dependent. This environment variable enables or disables dynamic reduction of the number of threads available for execution of parallel regions.
setenv OMP_NESTED *logical-value*

Here, *logical-value* is either *true* or *false*. The default setting is implementation dependent.
setenv OMP_SCHEDULE value

Here, value can be one of:
static[,chunksize]
dynamic
guided[,chunksize]
runtime

The default setting is implementation-dependent.
setenv OMP_NUM_THREADS *num-threads*

OMP_NUM_THREADS is used to specify the number of threads to be used in a parallel region. It is equivalent to the library function omp_set_num_threads with the latter taking precedent.
OpenMP Code Compilation on AIX

**Code Compilation**

*For Fortran 90:*
```bash
twister% xlf90_r single.f90 -qsuffix=f90 -O5 -qsmp=omp
```

*For C:*
```bash
twister% xlc_r single.c -O5 -qsmp=omp
```

**Code Execution**

```bash
twister% setenv OMP_NUM_THREADS 4
twister% a.out
```
We have several compilers: Gnu, Portland Group and Intel. Here is an example using the Intel compiler. Please go to BU’s Linux Cluster Repository (http://scv.bu.edu/SCV/Archive/linux-cluster) for information on Gnu and PGI compilers.

**Code Compilation**

*For Fortran 90:*

```
$ skate% ifc single.f90 -O3 -openmp
```

*For C:*

```
$ skate% icc single.c -O3 -openmp
```

**Code Execution**

```
$ skate% setenv OMP_NUM_THREADS 4
$ skate% a.out
```
References

OpenMP official site:
• The OpenMP Forum (http://www.openmp.org/)

Book:
• Parallel Programming in OpenMP
  Chandra, Dagum, et. al., Morgan Kaufmann Publishers

Tutorials:
Introduction to OpenMP – Boston University (concise; talking head)
(http://scv.bu.edu/SCV/Tutorials/OpenMP)
Introduction to OpenMP – Alliance (more comprehensive)
(http://pacont.ncsa.uiuc.edu:8900)

To download this presentation and associated sample codes :
http://scv.bu.edu/SCV/Archive/IBM/TUTORIALS.html or
http://scv.bu.edu/SCV/Archive/linux-cluster/TUTORIALS.html