Parallel Programming Using OpenMP

David Porter and Shuxia Zhang

Phone: (612) 626 0802 (help)
Email: help@msi.umn.edu

June 28, 2011
OpenMP is a parallel programming interface for shared memory architectures and is available on the Elmo, IBM Blade center, and the SGI Altix. For better use of OpenMP programming in high performance computing, the Supercomputing Institute will have a one-day workshop addressing the different aspects of OpenMP, such as parallel and worksharing constructs, data scope attribute clauses, and synchronization constructs.

A hands-on practice will follow the lecture. Provided examples will enable the users to insert OpenMP directives for different parallel tasks and data scope attribute clauses. The users are also encouraged to bring in their serial application codes. The User Support Staff members will help you parallelize the code with OpenMP.

Level: Introductory
Prerequisites: Knowledge of Fortran, C, or C++
10:00-10:30  Introduction to shared Memory Machines
10:30-11:15  Introduction to OpenMP and Worksharing Constructs
11:15-12:00  Hands on
12:00-1:00   Lunch
1:00-1:45    Data Scope Attributes
1:45-2:15    Hands on
2:15-2:55    Synchronization Constructs
2:55-3:00    Break
3:00-3:15    OpenMP2: FORTRAN
3:15-4:00    Hands on
Shared memory architectures at the Institute
Calhoun (SGI Altix XE 1300)

- 256 compute nodes
- Each node has 2 quad-core 2.66 GHz Intel Clovertown processors
- Total of 2048 cores
- 16 GB of memory per node
- Aggregate of 4.1 TB of RAM
- Three Altix 240 head nodes
- Diskless Boot Nodes
- Four Altix 240 node
- Infiniband 4x DDR HCA

Note: The word core refers to an independent processing element that is physically on the same chip with one or more other independent processing elements.

- Up to 8 OpenMP threads

http://www.msi.umn.edu/hardware/calhoun
HP Linux Cluster

- 1091 compute nodes
- Each node has 2 quad-core 2.8 GHz Intel Nehalem processors
- Total of 8,728 cores
- 24 GB of memory per node
- Aggregate of 26 TB of RAM

- QDR Infiniband interconnect

- Scratch space: Lustre shared file system
- Currently 128 TB

http://www.msi.umn.edu/Itasca
• A Sun Fire X4600 Linux cluster
  • Six computing nodes
    – Each of the computing nodes has 8 AMD Opteron 8356 processors sharing memory of 128 GB.
    – Each of the 8356 processors has four 2.3GHz cores with 512KB L2 cache.
    – Total 32 cores with memory of 4GB/Core
  • One interactive node
    – Four dual-core 3.0GHz AMD Opteron model 8222 processors. Each core has 1MB L2 cache.
    – 32GB main memory
Elmo

- **Network**
  All of the systems within Elmo are interconnected with Gigabit ethernet

- **Home Directories and Disks**
  800 GB of file space for home directories.
  Default quota per account is 5 GB.
  Back up is done nightly and kept for one month

- **Scratch Spaces**
  1 TB of file space allocated to /scratch1 file system.
  400GB per node for local /scratch space.
  Default quota per account is 50 GB. No Back up.
Koronis

- NIH
- **uv1000**
  Production system: 1152 cores, 3 TiB memory
- **Two uv100s**
  Development systems: 72 cores, 48 GB, TESLA
- **One uv10 and three SGI C1103 sysems**
  Interactive Graphics nodes

static.msi.umn.edu/tutorial/hardwareprogramming/Koronis_2011june16_final.pdf

www.msi.umn.edu/hardware/koronis
UV1000: ccNUMA Architecture

ccNUMA:
- Cache coherent non-uniform memory access
- Memory local to processor but available to all
- Copies of memory cached locally

NUMAlink 5 (NL5)
- SGI’s 5th generation NUMA interconnect
- 4 NUMAlink 5 lines per processor board
- 7.5 GB/s (unidirectional) peak per NL5 line
- 2-D torus of NL5 lines between boardpairs
OpenMP
Outline

• What is OpenMP?
• Constructs (Directives and Clauses)
  - Control
    - Parallel region
    - Work-sharing
    - Combined parallel work-sharing
  - Data environment Construct
    (Data Scope attribute Clauses)
  - Synchronization constructs
• Run-time library routines and Environment variables
• OpenMP 2
What is OpenMP?

- An API
- Fork-Join model of parallel execution

- Execution starts with one thread—Master thread
- Parallel regions fork off new threads on entry—Team thread
- Threads join back together at the end of the region - only master thread continues
• Model for parallel programming
• Portable across shared-memory architectures
• Scalable
• Compiler based
  - Most OpenMP constructs are compiler directives or pragmas
• Extensions to existing programming languages
  - Mainly by directives
  - A few library routines
• Fortran and C/C++ binding
• Supports data parallelism
• OpenMP is a shared memory model

• 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:
  - Race condition: program’s outcome changes when threads are scheduled differently
• OpenMP has three primary components
  – Compiler Directives
  – Runtime Library Routines
  – Environment Variables

• Portable and standardized
  – Fortran, C, and C++ Specifications
  – Definition by Parallel Computing Forum.
    Defined and endorsed by hardware and software vendors.

• OpenMP support at the Institute's computer platforms
  – Intel compilers
  – Fortran, C and C++ on a node
Compiling and Running OpenMP

- To compile (Intel Compiler):
  - Fortran: `ifort -O3 -openmp program.f`
  - C: `icc -O3 -openmp program.c`

- To run:
  - Interactive: `export OMP_NUM_THREADS=4
  ./a.out`

  • Batch: Use PBS
    `www.msi.umn.edu/hardware/elmo/quickstart.html`
• **OpenMP directive format**
  – Fortran
    
    !$OMP directive_name clauses
    !$OMP, *$OMP, C$OMP
  – C
  
    #pragma omp directive_name clauses

• **Automatic Parallelization**
° Shared-memory architectures

Shared Memory

° Each CPU can read & write all of the memory
CPUs can only see memory on their own node & need to pass messages (MPI) to communicate with other nodes.
OpenMP Worksharing Constructs
Parallel Region

° Parallel directives: simple & few in number

° Parallel region defined by
  - PARALLEL / END PARALLEL
    - Fundamental: does the actual fork and join parallel execution
    - Number of threads won't change inside a parallel region
    - Single Program Multiple Data (SPMD) execution within region
    - Pair must appear in the same routine
    - No branching into or out of block
    - More on clauses (data environment) later
    - Format

  Fortran:  !$OMP PARALLEL [clause[,] clause]...
            block
            !$OMP END PARALLEL

  C/C++:    #pragma omp parallel [clause] newline
            Structured_ block
Parallel Loop

° Work-sharing
° DO / END DO
  - The classic parallel loop
  - Must be inside a parallel region
  - Iterations distributed across existing threads
  - Loop index is private to thread by default
  - Loop index must be of type INTEGER
  - If used, the END DO must appear immediately after the loop
  - Branching out of loop is illegal
  - More on clauses (data environment) later
  - Format

  Fortran:  !$OMP DO [clause[..] clause]...
           do_loop
           !$OMP END DO [NOWAIT]

  C/C++:  #pragma omp for [clause] newline
          for_loop
Example

real a(36), b(36), c(36)
! Initialize a, b, & c
...
!$omp parallel shared(a,b,c), private(i)
!$omp do
  do i=1, 36
    a(i) = b(i)+c(i)
  enddo
!$omp end do nowait
!$omp end parallel
Parallel Sections

° **SECTIONS / END SECTIONS**

° Non-iterative work-sharing
° Enclosed sections divided among threads
° Must be inside a parallel region
° Each section is executed once by a thread
° Format:

Fortran

```
!$OMP SECTIONS [clause[[,] clause]...] nowait
  !$OMP SECTION
    block
  !$OMP SECTION
    block
  ....
!$OMP END SECTIONS [NOWAIT]
```

C

```
#pragma omp sections
{
  #pragma omp section
    structured_block
  #pragma omp section
    structured_block
}
```
Parallel Sections

° **SECTIONS / END SECTIONS** cont’d
  ° **SECTION** directives: must be within the lexical extent of **SECTIONS / END SECTIONS**
  ° Illegal to branch into or out of constituent section (**SECTION**) blocks
  ° Illegal to branch into or out of code enclosed by **SECTIONS / END SECTIONS**
  ° More on clauses (data environment) later
Example

real a(36), b(36), c(36)
!$omp parallel shared(a,b,c), private(i)
!$omp sections
!$omp section
do 10 i=1,36
10   a(i) = ...
!$omp section
do 20 i=1,36
20   b(i) = ...
!$omp section
do i=1,36
30   c(i) = ...
!$omp end sections
!$omp end parallel
SINGLE / END SINGLE

- Encloses code to be executed by only one thread
- Useful for (short) sequential section within the parallel region
- Illegal to branch into or out of code enclosed by SINGLE / END SINGLE
- More on clauses (data environment) later

Format:

Fortran:

```fortran
$OMP SINGLE [clause[, clause]...] block
$OMP END SINGLE [NOWAIT]
```

C:

```c
#pragma omp single [clause ...] newline
structured_block
```

Example

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

```cpp
!$OMP PARALLEL
  CALL S1
!$OMP SINGLE
    CALL S2
!$OMP END SINGLE
  CALL S3
!$OMP END PARALLEL
```
° **MASTER / END MASTER**
  ° SINGLE on master thread
  ° However, no implied barrier on entry or exit
  ° Illegal to branch in or out

° **Format:**

  Fortran:
  
  ```fortran
  !$OMP MASTER
  block
  !$OMP END MASTER
  ```

  C:
  
  ```c
  #pragma omp master newline
  structured_block
  ```
° Combined parallel work-sharing
  ° PARALLEL DO / END PARALLEL DO
    ° Convenient combination of PARALLEL and DO for a parallel region that contains a single DO directive
    ° Semantics identical to explicitly specifying PARALLEL followed immediately by DO
    ° Accepts any of the clauses for PARALLEL or DO directive
    ° If used, the END PARALLEL DO must appear immediately after the loop

° Format:
  Fortran:  !$OMP PARALLEL DO [clause[[],] clause]…] 
            block
            !$OMP END PARALLEL DO 

  C/C++:    #pragma omp parallel [clause] newline 
            Structured_ block
PARALLEL SECTIONS / END PARALLEL SECTIONS

- Convenient combination of PARALLEL and SECTIONS for a parallel region that contains a single SECTIONS directive
- Semantics identical to explicitly specifying PARALLEL followed immediately by SECTIONS
- Accepts any of the clauses for PARALLEL or SECTIONS directive

Format:

Fortran:

```fortran
 !$OMP PARALLEL SECTIONS [clause[,] clause]…
 Block
 !$OMP END PARALLEL SECTIONS
```

C/C++:

```c
#pragma omp parallel sections [ clause …] newline
 Structured_block
```
Login to SDVL

Problems posted at:
http://static.msi.umn.edu/tutorial/scicomp/general/openMP/workshop_OpenMP

To Compile:
module load intel
ifort –O3 –openmp yourcode.f
icc –O3 –openmp yourcode.c

To run:
export OMP_NUM_THREADS=2
/usr/bin/time ./a.out
OpenMP Data Scope attributes
Clauses
Introduction

- Several directives accept clauses (key words) that allow a user to control the scope attributes of variables.
- Not all clauses are allowed on all directives, but the clauses that are valid on a particular directive are described.
- If no data scope clauses are specified for a directive, the default scope for variables affected by the directive is `shared` unless you set the default to `private` or `none`.
- The effective execution of clauses is case-sensitive in C program, but not in FORTRAN.
- The order of specifying a few clauses does not affect the execution.
**default clause**

The **default** clause lets you specify a scope for all variables in the lexical extent of a parallel region. Syntax:

FORTRAN:  
```
!$OMP  default (shared)
```

C:  
```
#pragma omp default (shared)
```

or

```
!$OMP   default (private)
```

```
#pragma omp default (private)
```

or

```
!$OMP  default (none)
```

```
#pragma omp default (none)
```
**Data scope clauses**

**default clause**

- **private** - Makes all named objects in the lexical extent of the parallel region, including common block variables but excluding **threadprivate** variables.

- **shared** - Makes all named objects in the lexical extent of the parallel region shared among the threads in a team, as if you explicitly listed each variable in a shared clause. If you do not specify a default clause, this is the default.

- **none** - Specifies that there is no implicit default as to whether variables are **private** or **shared**. In this case, you must specify the **private, shared, firstprivate, lastprivate** or **reduction** property for each variable you use in the lexical extent of the parallel region.

- **default** clause defined only in a **parallel** directive. You can exclude variables from a defined default by using the **private, shared, firstprivate, lastprivate** or **reduction** clauses. Variables in **threadprivate** common blocks are not affected by the **default** clause.
private clause  FORTRAN:  !$OMP  private(list)
               C:             #pragma omp private(list)

where list is the name of one or more variables or common blocks that are
accessible to the scoping unit. Each name must be separated by a comma,
and a named common block must appear between slashes (/ /)

The variables specified in a private list are private to each thread. When an
assignment to a private variable occurs, each thread assigns to its local copy
of the variable. When operations involving a private variable occur, each
thread performs the operations using its local copy of the variable.

Variables declared private in a parallel region are undefined upon entry to the
parallel region. If the first use of a private variable within the parallel region is
in a right-hand-side expression, the results of the expression will be undefined
(i.e. this is probably a coding error).

Likewise, variables declared private in a parallel region are undefined when
serial execution resumes at the end of the parallel region.
The **shared** clause specifies variables that will be shared by all the threads in a team, meaning that all threads access the same storage area for **shared** data. Syntax:

**FORTRAN:**

```fortran
$OMP shared (list)
```

**C:**

```c
#pragma omp shared (list)
```

where list is the name of one or more variables or common blocks that are accessible to the scoping unit. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).
firstprivate clause

The firstprivate clause provides a superset of the functionality provided by the private clause so that they are initialized with certain values.

Syntax:

FORTRAN: 
$\texttt{!$OMP firstprivate (list)}$

C: 
$\texttt{#pragma omp firstprivate (list)}$

where list is the name of one or more variables or common blocks that are accessible to the scoping unit. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables that appear in a firstprivate list are subject to private clause semantics. In addition, private (local) copies of each variable in the different threads are initialized to the value the variable had before the parallel region started.
Example: firstprivate

```fortran
real*8 a(100,100), b(100,100), c(100)
integer n,i
n=100
m=100
do i=1,n
  c(i)=i*100.
do j=1,m
  b(i,j)=(i-1)*m/float(m+n)
end do
end do

!$omp parallel do private (i,j)
!$omp& shared (a,b,m,n) firstprivate(c)
do j=1,n
  do i=2,m-1
    c(i)=sqrt(1.0+b(i,j)**2)
  end do
  do i=1,n
    a(i,j)=sqrt(b(i,j)**2+c(i)**2)
  end do
end do
!$omp end parallel do

do i=1,10
  print *, 'i=',i, ' a(i,5) ', a(i,5)
end do
print *, '....'
print *, '....'
do i=1,10
  print *, 'i=',i+90, ' a(i,5) ', a(i+90,5)
end do
end
```
Example: first private

```
i=  1  a(i,5)  100.000000000000000
i=  2  a(i,5)  1.22474487139158916
i=  3  a(i,5)  1.73205080756887742
i=  4  a(i,5)  2.34520787991171487
i=  5  a(i,5)  3.00000000000000000
i=  6  a(i,5)  3.67423461417476727
i=  7  a(i,5)  4.35889894354067398
i=  8  a(i,5)  5.04975246918103871
i=  9  a(i,5)  5.74456264653802862
i= 10  a(i,5)  6.4420493636256306
....
i= 91  a(i,5)  63.6474665638782540
i= 92  a(i,5)  64.3544870230506945
i= 93  a(i,5)  65.0615093584524828
i= 94  a(i,5)  65.7685335095743113
i= 95  a(i,5)  66.4755594184810121
i= 96  a(i,5)  67.1825870296760712
i= 97  a(i,5)  67.8896162899747111
i= 98  a(i,5)  68.5966471483847329
i= 99  a(i,5)  69.303679559947194
i= 100 a(i,5)  10000.1225117495433```
The `lastprivate` clause provides a superset of the functionality provided by the `private` clause; objects are declared `private` and they are given certain values when the parallel region is exited.

**FORTRAN:**

```fortran
!$OMP lastprivate (list)
```

**C:**

```c
#pragma omp lastprivate (list)
```

where `list` is the name of one or more variables or common blocks that are accessible to the scoping unit. Each name must be separated by a comma, and a named common block must appear between slashes (`/ `/).

Variables that appear in `lastprivate` list are subject to `private` clause semantics. In addition, once the parallel region is exited, each variable has the value provided by the sequentially last section or loop iteration.
Example: Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a lastprivate clause so that the values of the variables are the same as when the loop is executed sequentially.

```fortran
$OMP PARALLEL
$OMP DO LASTPRIVATE(I)
   DO I=1,N
      A(I) = B(I) + C(I)
   ENDDO
$OMP END PARALLEL
CALL REVERSE(I)
```

In the preceding example, the value of I at the end of the parallel region will equal N+1, as in the sequential case.
threadprivate clause

The threadprivate directive specifies named common blocks or file-scope to be private (local) to a thread; they are global within the thread.

FORTRAN:    !$omp threadprivate( /cb/ [, /cb/]...)  
C:    #pragma omp threadprivate(cb)

where cb is the name of the common block you want made private to a thread. Only named common blocks can be made thread private.
Rules:

Each thread gets its own copy of the common block. During serial portions and MASTER sections of the program, accesses are to the master thread copy of the common block. On entry to the first parallel region, data in the threadprivate common blocks should be assumed to be undefined unless a copyin clause is specified in the parallel directive.

A threadprivate common block or its constituent variables can appear only in a copyin clause. They are not permitted in a private, firstprivate, lastprivate, shared, or reduction clauses. They are not affected by the default clause.
Examples: In the following example, the common blocks BLK and FIELDS are specified as thread private:

```
COMMON /BLK/ SCRATCH
COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK/, /FIELDS/)
!$OMP PARALLEL DEFAULT(PRIVATE) COPYIN(/BLK/, ZFIELD)
```
Reduction

reduction clause

The reduction clause performs a commutative reduction operation on the specified variables. Syntax:

FORTRAN:   !$OMP  reduction (operator/intrinsic : list )
C:           #pragma omp reduction (operator/intrinsic :list )

where operator is one of the following: +, *, -, .AND., .OR., .EQV., .or., .NEQV., and intrinsic is one of the following: MAX, MIN, IAND, IOR, or IEOR.

Variables in list must be named scalar variables of intrinsic type. There is no guarantee that bit-identical results will be obtained for floating point reductions from one parallel run to another. Variables appeared in a reduction clause must be shared in the enclosing context. Any number of reduction clauses can be specified on the directive, but a variable can appear only once for that directive.
The following table lists the operators and intrinsics that are valid and their canonical initialization values. The actual initialization value will be consistent with the data type of the reduction variable.

**Table: Initialization Values for reduction computation**

<table>
<thead>
<tr>
<th>Operator/Intrinsic</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>.AND.</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.OR.</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>.EQV.</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>MAX</td>
<td>Smallest representable number</td>
</tr>
<tr>
<td>MIN</td>
<td>Largest representable number</td>
</tr>
<tr>
<td>IAND</td>
<td>All bits on</td>
</tr>
<tr>
<td>IOR</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: How to use the reduction clause:

!$OMP PARALLEL DO DEFAULT(PRIVATE)
!$OMP& SHARED(N) REDUCTION(+: A,B)
  DO I=1,N
    CALL WORK(ALOCAL,BLOCAL)
    A = A + ALOCAL
    B = B + BLOCAL
  ENDDO
!$OMP END PARALLEL DO
The schedule clause controls how the iterations of the loop are assigned to threads.

**static:** Each thread is given a chunk of iterations in a round robin order. Least overhead - determined statically

**dynamic:** Each thread is given chunk iterations at a time; more chunks distributed as threads finish Good for load balancing

**guided:** Similar to dynamic, but chunk size is reduced exponentially

**runtime:** User chooses at runtime using environment variable

For example: export OMP_SCHEDULE="dynamic,4"
export OMP_SCHEDULE="static,10"
export OMP_SCHEDULE="guided,2"

The runtime setup will override what is defined in the code.
schedule

!$OMP PARALLEL DO &
!$OMP SCHEDULE (STATIC, 3)
DO J = 1, 36
  Work (j)
END DO
!$OMP END DO

!$OMP PARALLEL DO &
!$OMP SCHEDULE (DYNAMIC, 1)
DO J = 1, 36
  Work (j)
END DO
!$OMP END DO

!$OMP PARALLEL DO &
!$OMP SCHEDULE (GUIDED, 1)
DO J = 1, 36
  Work (j)
END DO
!$OMP END DO
Hands On

Login to SDVL

Problems posted at:
http://static.msi.umn.edu/tutorial/scicomp/general/openMP/workshop_OpenMP

To Compile:
   module load intel
   ifort –O3 –openmp yourcode.f
   icc –O3 –openmp yourcode.c

To run:
   export OMP_NUM_THREADS=2
   /usr/bin/time ./a.out

   ssh login.msi.umn.edu
   isub –n nodes=1:ppn=4 –m 8gb
OpenMP Synchronization
Synchronization directives overview

- **Implicit barriers** (wait for all threads)
  - DO / END
  - PARALLEL DO / END PARALLEL DO
  - SECTIONS / END SECTIONS
  - PARALLEL SECTIONS / END PARALLEL SECTIONS
  - SINGLE / END SINGLE
    - Note: For MASTER / END MASTER
    - no implied barrier
  - NOWAIT at END
    - overrides implicit synchronization

```c
!$OMP PARALLEL
!$OMP DO
  DO I=2, N
    B(I) = (A(I) + A(I-1)) / 2.0
  ENDDO
!$OMP END DO NOWAIT
!$OMP DO
  DO I=1, M
    Y(I) = SQRT (Z(I))
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
```
Explicit synchronization directives

**BARRIER**

This directive synchronizes all the threads in a team. When encountered, each thread waits until all of the others threads in that team have reached this point.

**CRITICAL** 

The **CRITICAL** and **END CRITICAL** directives restrict access to the enclosed code to only one thread at a time. The optional *name* argument identifies the critical section. It is illegal to branch into or out of **CRITICAL** code section. If *name* is specified in **CRITICAL**, same *name* must be specified in **END CRITICAL**

- **MUTEX**
Example

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)
!$OMP CRITICAL (XAXIS)
  CALL DEQUEUE(IX_NEXT, X)
!$OMP END CRITICAL (XAXIS)
  CALL WORK(IX_NEXT, X)

!$OMP CRITICAL (YAXIS)
  CALL DEQUEUE(IY_NEXT, Y)
!$OMP END CRITICAL (YAXIS)
  CALL WORK(IY_NEXT, Y)

!OMP END PARALLEL
**Atomic directive**

**ATOMIC**
- Single-statement critical section for reduction
- Applies to the immediately following statement which may be of the form
  
  \[
  x = x \text{ operator } \text{expr} \quad \text{OR} \quad x = \text{expr} \text{ operator } x \\
  x = \text{intrinsic}(x, \text{expr}) \quad \text{OR} \quad x = \text{intrinsic}(x, \text{expr})
  \]

  - **ATOMIC** directive ensures that load/store operations in the specified statement are executed one thread at a time (atomically). The functionality is similar to that of **CRITICAL**, but applies only to the immediately following statement.

```fortran
!$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X, Y, INDEX, N)
DO I= 1, N
   CALL WORK (XLOCAL, YLOCAL)
!$OMP ATOMIC
   X(INDEX(I)) = X(INDEX(I)) + XLOCAL
   Y(I) = Y(I) + YLOCAL
ENDO
!$OMP END PARALLEL DO
```
### FLUSH [(list)]

- Synchronization point at which the implementation is required to provide a consistent view of memory
- Must appear at the precise point where needed
- Optional argument `list`: comma-separated variables that need to be flushed
- If `list` is not specified, all thread-visible variables (global, dummy arguments, pointer dereferences, shared local) are flushed

```c
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED (ISYNC)
IAM = OMP_GET_THREAD_NUM()
ISYNC(IAM) = 0
!$OMP BARRIER
CALL WORK()
C I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
ISYNC(IAM) = 1
!$OMP FLUSH
C WAIT TILL NEIGHBOR IS DONE
   DO WHILE (ISYNC(NEIGH). EQ. 0)
!$OMP FLUSH(ISYNC)
ENDDO
!$OMP END PARALLEL
```
Ordered directive

**ORDERED / END ORDERED**

- For pipelining loop iterations
- Can exist only in the dynamic extent of a `DO` or `PARALLEL DO` directive
- The `DO` directive to which it binds must have the `ORDERED` clause specified
- Only one thread can enter at a time
- It is illegal to branch into or out of `ORDERED` code section

```plaintext
!$OMP DO ORDERED SCHEDULE(DYNAMIC)
   DO I=LB,UB,ST
       CALL WORK(I)
   ENDDO
!$OMP END DO
...
   SUBROUTINE WORK(K)
   !$OMP ORDERED
   WRITE(*,*) K
   !$OMP END ORDERED
   return
end
```
OpenMP Environment & Runtime Library
**OpenMP Environment & Runtime Library**

- For controlling execution
  - Needed for tuning, but may limit portability
  - Control through environment variables or runtime library calls
    - Runtime library takes precedence in conflict
• **OMP_NUM_THREADS**: How many to use in parallel region
  - `OMP_GET_NUM_THREADS, OMP_SET_NUM_THREADS`
  - Related: `OMP_GET_THREAD_NUM, OMP_GET_MAX_THREADS, OMP_GET_NUM_PROCS`

• **OMP_DYNAMIC**: Should runtime system choose number of threads? (TRUE or FALSE)
  - `OMP_GET_DYNAMIC, OMP_SET_DYNAMIC`

• **OMP_NESTED**: Should nested parallel regions be supported?
  - `OMP_GET_NESTED, OMP_SET_NESTED`

• **OMP_SCHEDULE**: Choose DO scheduling option
  - Used by `RUNTIME` clause

• **OMP_IN_PARALLEL()**: Is the program in a parallel region?
  - Returns `.TRUE.` or `.FALSE.`
• Nested Parallelism
  - Requires creating new parallel region
  - Not supported on all OpenMP implementations

• Orphaned directive
  - An OpenMP directive which appears outside of the static (lexical) extent of a parallel region
  - Example: code in a called subroutine
OpenMP2
OpenMP FORTRAN Application Program Interface
Version 2.0

Major new features:

• COPYPRIVATE for broadcast of sequential reads
• Parallelization of F90 array syntax
• Privatization of module data
• Array reductions
• Portable timing routines
• Control of the number of threads for multi-level parallelism
FORTRAN Support

Parallelization of F90 array syntax via the WORKSHARE directive

The FORTRAN 77 standard does not require that initialized data have the SAVE attribute but Fortran 90 and 95 does require this. OpenMP Fortran version 2.0 requires this.
COPYPRIVATE

The COPYPRIVATE clause uses a private variable to broadcast a value from one member of a team to the other members. The COPYPRIVATE clause can only appear on the END SINGLE directive.

Example:

```
INTEGER I
!$OMP PARALLEL PRIVATE (I)
...
!$OMP SINGLE READ (*, *) !
!$OMP END SINGLE COPYPRIVATE (I)
! In all threads in the team, I is equal to
! the value that you entered. !
...
!$OMP END PARALLEL
```
WORKSHARE directive

Allows the parallelization of F90 array expressions.

Syntax:

    !$OMP WORKSHARE [clause[,] clause]...
    block
    !$OMP END WORKSHARE [NOWAIT]

Where block is a piece or full array satisfying F90 syntax, like MATMUL, DOT_PRODUCT, SUM, PRODUCT, MAXVAL, MINVAL, RESHAPE, TRANSPOSE, etc.

A BARRIER is implied following the enclosed code if the NOWAIT clause is not specified on the END WORKSHARE directive.
WORKSHARE directive

Role:

Directive binds to the closest dynamically enclosing PARALLEL directive

NOT nest DO, SECTIONS, SINGLE and WORKSHARE directives that bind to the same PARALLEL directive

NOT specify a WORKSHARE directive within CRITICAL, MASTER, or ORDERED directives.

NOT specify BARRIER, MASTER, or ORDERED directives within the dynamic extent of a WORKSHARE construct.

A BARRIER directive is implied at the END unless a NOWAIT is specified.

A WORKSHARE construct must be encountered by all threads in the team or by none at all.
WORKSHARE directive

Example:

$OMP WORKSHARE
FORALL (I = 1 : N, AA(1, I) = 0)
AA(1, I) = I
BB = TRANSPOSE(AA)
CC = MATMUL(AA, BB)
!$OMP ATOMIC
S = S + SUM(CC)
!$OMP END WORKSHARE .
Portable Wallclock timers:

The OpenMP run-time library includes two routines supporting a portable wall-clock timer.

```
DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
```

Example:

```
DOUBLE PRECISION START, END
START = OMP_GET_WTIME()
!.... work to be timed
END = OMP_GET_WTIME()
PRINT *, 'Stuff took ', END-START,' seconds'
```
NUM_THREADS

NUM_THREADS clause allows the dynamic spawning of threads

Example:

```
DIMENSION X(1000,500)
!$OMP PARALLEL WORKSHARE SHARED(X,Y), NUMBER_THREADS(4)
   X=100
!$OMP END PARALLEL WORKSHARE
```

A specific number of threads is used in a parallel region. It supersedes the number of threads indicated by the OMP_SET_NUM_THREADS or the OMP_NUM_THREADS environment variable for the parallel region it is applied to.
Extension of THREADPRIVATE and COPYIN

THREADPRIVATE may now be applied to variables as well as COMMON blocks.
COPYIN now works on variables as well as COMMON blocks.
Reprivatization of variables is now allowed in OpenMP 2
Hands On

Login to SDVL

Problems posted at:
http://static.msi.umn.edu/tutorial/scicomp/general/openMP/workshop_OpenMP

To Compile:
  module load intel
  ifort -O3 -openmp yourcode.f
  icc -O3 -openmp yourcode.c

To run:
  export OMP_NUM_THREADS=2
  /usr/bin/time ./a.out
User Support:
E-mail: help@msi.umn.edu
Phone: (612) 626-0806
Webpage:
http://www.msi.umn.edu