Shared Memory Programming With OpenMP

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2014

Specifications

All specs and API descriptions can be found at:

`http://www.openmp.org`

This presentation covers

1. Unified 4.0 API (2013-07) [Highlights]
2. Unified 3.1 API (2011-07)

History

Brief history of OpenMP:

- 1997-10 `FORTRAN` 1.0
- 1998-10 `C/C++` 1.0
- 1999-11 `FORTRAN` 1.1
- 2000-11 `FORTRAN` 2.0
- 2002-03 `C/C++` 2.0
- 2005-05 Unified `FORTRAN/C/C++` 2.5
- 2008-05 Unified API 3.0
- 2011-07 Unified API 3.1
- 2013-07 Unified API 4.0

What is OpenMP?

- **Open** Specifications for **M**ulti **P**rocessing
- An Application Programming Interface (API) intended for directing multi-threaded shared memory parallelism:
  - Compiler directives
  - Run-time library routines
  - Environmental variables
- Portable - specified for `C/C++` and `FORTRAN` (requires OpenMP compliant compiler)
- Standard - joint effort by major hardware and software vendors (not yet an ANSI standard)

OpenMP Strengths

- Ease of (mis)use
- Incremental parallelization
- Fairly easy to get speedup
- Potentially scalable on large (SMP) systems
- HPC architectures are evolving towards OpenMP's design(Multi-core, single socket CPUs are already here, and gaining rapidly in popularity)

OpenMP Weaknesses

OpenMP is **not**

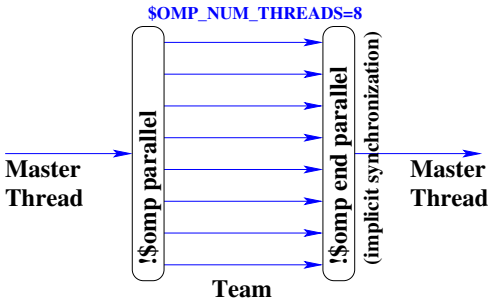
- intended for distributed memory parallel computing (can be used in combination with MPI, however)
  - Intel's Cluster OpenMP, extend over distributed memory
  - UPC, Co-Array `FORTRAN`, proposed PGAS language extensions for shared & distributed memory
- implemented identically by all vendors (not a surprise)
- promised to be the most efficient way to make use of shared memory (data locality is still an outstanding issue)

OpenMP Design Goals

- Thread-based: a shared memory **process** can consist of multiple threads - OpenMP is based on the idea of controlling and using these threads
- Explicitly parallel: **not** automatic - OpenMP allows the programmer **full** control over parallelization
- Compiler directive-based: most OpenMP parallelism is controlled through the use of compiler directives
- Dynamic threads: the number of threads can be dynamically changed for different parallel regions

- Nested support: parallel regions can be nested, but this feature is left as implementation dependent OpenMP 3.0 clarifies nesting and includes new routines and control variables for nesting parallel regions.
- Fork-join model: the master thread (originating process) spawns a team of parallel threads on encountering the first parallel region. The threads synchronize and terminate at the end of the parallel region construct

Execution Model



- *Fork-Join*, the master thread spawns a team of threads inside parallel regions.
- *Typical Use*: split compute-intensive loops among the thread team. See the previous slide.

OpenMP General Syntax

Most OpenMP constructs are compiler directives or pragmas (we will deal with the OpenMP API separately):

C/C++:

```
1 #pragma omp construct[clause[clause]...]
```

F77:

```
1 C$OMP construct[clause[clause]...]
```

F90:

```
1 !$OMP construct[clause[clause]...]
```

to compilers that do not support OpenMP, these directives are comments, and have no effect.

Ease-of-use can come with a price...

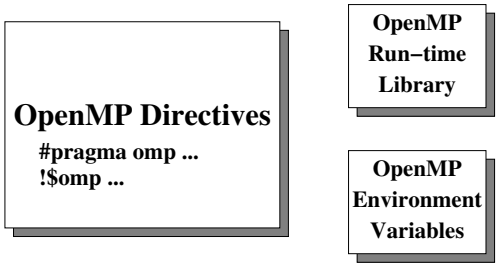
- OpenMP does not force the programmer to explicitly manage communication or how the program data is mapped onto individual processors - sounds great ...
- OpenMP program can easily run into common SMP programming errors, usually from resource contention issues.

Shared vs. Distributed

Compare shared memory (SM, OpenMP) versus distributed memory (DM, MPI) in the following table of features:

Feature	SM	DM
Parallelize subsections of application	Pretty easy; often significant reward for small investment	Pretty difficult (often have to rework application)
Scalability on large processor counts	Few such systems (large ccNUMA)	Clusters with high performance interconnects very common
Complexity over serial code	Simple algorithms easy, more complex ones can be involved	Complex even in simple cases
Code maintainability	Directives in otherwise serial code - minimal in many cases	Message handling requires significant overhead

Components of OpenMP



We will consider the smaller pieces of the OpenMP puzzle first (they are reasonable self-contained and will help to inform the rest)

OpenMP References

- Book: "Using OpenMP: Portable Shared Memory Parallel Programming," by B. Chapman, G. Jost, and R. van der Pas (MIT, Boston, 2008).
  - Sample codes available online at `www.openmp.org`
- Book: "Parallel Programming in OpenMP," by R. Chandra et. al. (Academic, New York, 2001).
- Web: `www.openmp.org`

Categories

The OpenMP API is relatively small. The API provides routines in several categories:

- Control and query the parallel execution environment
  - Monitor threads and processors
  - Allow dynamic thread adjustments
  - Enable nested parallelism
- Lock functions to be used to serialize/synchronize data access
  - Initialize, test, remove simple and nestable locks

Accessing the Run-time Library Functions

Syntax for setting the number of OpenMP threads:

in C/C++ :

```
1 #include <omp.h>
2 ...
3 void omp_set_num_threads(int num_threads);
```

in FORTRAN 77 :

```
1 include "omp_lib.h"
2 ...
3 call omp_set_num_threads(num_threads);
```

in FORTRAN 90 :

```
1 USE omp_lib
2 ...
3 call omp_set_num_threads(num_threads);
```

Environmental Functions

FORTRAN	C/C++
subroutine OMP_SET_NUM_THREADS	void omp_set_num_threads(int nthreads)
integer function OMP_GET_NUM_THREADS	int omp_get_num_threads(void)
integer function OMP_GET_MAX_THREADS	int omp_get_max_threads(void)
integer function OMP_GET_THREAD_NUM	int omp_get_thread_num(void)
integer function OMP_GET_NUM_PROCS	int omp_get_num_procs(void)
logical function OMP_IN_PARALLEL	int omp_in_parallel(void)
subroutine OMP_SET_DYNAMIC(scalar_logical_expr)	void omp_set_dynamic(int dthreads)
logical function OMP_GET_DYNAMIC	int omp_get_dynamic(void)
subroutine OMP_SET_NESTED(scalar_logical_expr)	void omp_set_nested(int nested)
logical function OMP_GET_NESTED	int omp_get_nested(void)

Some new routines added in OpenMP 3.0 will be discussed later.

Run-time Locks Overview

simple locks may not be locked if already in a locked state.

nestable locks may be locked multiple times by the same thread.

Lock variables:

```
1 INTEGER(KIND=OMP_LOCK_KIND) :: svar
2 INTEGER(KIND=OMP_NEST_LOCK_KIND) :: nvar
```

```
1 omp_lock_t *lock;
2 omp_nest_lock_t *lock;
```


Lock Functions

FORTRAN	C/C++
subroutine OMP_INIT_LOCK(svar)	void omp_init_lock(omp_lock_t *lock)
subroutine OMP_INIT_NEST_LOCK(nvar)	void omp_init_nest_lock(omp_nest_lock_t *lock)
subroutine OMP_DESTROY_LOCK(svar)	omp_destroy_lock(omp_lock_t *lock)
subroutine OMP_DESTROY_NEST_LOCK(nvar)	void omp_destroy_nest_lock(omp_nest_lock_t *lock)
subroutine OMP_SET_LOCK(svar)	void omp_set_lock(omp_lock_t *lock)
subroutine OMP_SET_NEST_LOCK(nvar)	void omp_set_nest_lock(omp_nest_lock_t *lock)
subroutine OMP_UNSET_LOCK(svar)	void omp_unset_lock(omp_lock_t *lock)
subroutine OMP_UNSET_NEST_LOCK(nvar)	void omp_unset_nest_lock(omp_nest_lock_t *lock)
logical function OMP_TEST_LOCK(svar)	int omp_test_lock(omp_lock_t *lock)
logical function OMP_TEST_NEST_LOCK(nvar)	int omp_test_nest_lock(omp_nest_lock_t *lock)

Simple Lock Example

It is easy to get into trouble with locks - setting a lock will cause the non-owning threads to block until the lock is unset, consider:

```
1  #include <omp.h>
2
3  omp_lock_t *lock1;
4
5  /* lots of intervening code ... */
6
7  omp_init_lock(lock1);
8
9  #pragma omp parallel for shared(lock1)
10 for (i=0; i<=N-1; i++) { /* simple lock example */
11   if (A[i] > CURRENT_MAX) {
12      omp_set_lock(&lock1);
13      if (A[i] > CURRENT_MAX) {
14         CURRENT_MAX = A[i];
15      }
16   }
17   omp_unset_lock(&lock1);
18 }
19 omp_destroy_lock(&lock1);
```

Similar serialization can also be obtained using directives, albeit not with the same level of control.

Timing Routines

The run-time library includes two timing routines that implement a portable wall-clock timer.

in FORTRAN :

```
1 double precision function OMP_GET_WTIME()
2 double precision function OMP_GET_WTICK()
3 [number of seconds between clock ticks]
```

in C/C++ :

```
1 double omp_get_wtime(void)
2 double omp_get_wtick(void)
```


Usage example:

in FORTRAN :

```
1       DOUBLE PRECISION tstart,tend
2       tstart=OMP_GET_WTIME()
3       call bigjob(i,j,M,N,a)
4       tend=OMP_GET_WTIME()
5       print*, 'bigjob exec time = ',tend-tstart
```

in C/C++ :

```
1 double tstart;
2 double tend;
3 tstart = omp_get_wtime();
4 ... work to be timed ...
5 tend = omp_get_wtime();
6 printf("Work took %f seconds\n", tend - tstart);
```


OpenMP Environmental Controls

OMP_NUM_THREADS integer
: how many default threads used in parallel regions

OMP_SCHEDULE (type[,chunk])
: control default for SCHEDULE directive, type can be one of **static, dynamic**, and**guided**. Also **auto** in OpenMP 3.0.

OMP_DYNAMIC true|false
: allows/disallows variable number of threads

OMP_NESTED true|false
: allows/disallows nested parallelism. If allowed, number of threads used to execute nested parallel regions is **implementation dependent** (can even be serialized!).

- Vendors may have additional env variables (e.g. Intel for data/thread placement or CPU affinity).

Environmental Examples

tcsh :

```
1 setenv OMP_NUM_THREADS 2
2 setenv OMP_SCHEDULE "guided,4"
3 setenv OMP_SCHEDULE "dynamic"
```

bash :

```
1 export OMP_NUM_THREADS=2
2 export OMP_SCHEDULE="guided,4"
3 export OMP_SCHEDULE="dynamic"
```

Directive Sentinels

in FORTRAN: Fixed-form source code, must start in column 1 with no intervening white space:

```
1 !$OMP
2 C$OMP
3 *$OMP
```

Free-form source uses just the first (can appear in any column as long as it is preceded only by white space):

```
1 !$OMP
```

in C/C++: (free-form source, of course)

```
1 #pragma omp directive-name [clause[[,]clause]...]
```


OpenMP Conditional Compilation Details

in FORTRAN: fixed-source form, conditional compilation sentinels must start in column 1

```
1 !$
2 C$
3 *$
4 c$
```

Examples:

```
1 !$ 10  IAM = OMP_GET_THREAD_NUM() +
2 !$   & INDEX
3 #ifdef _OPENMP
4    10 IAM = OMP_GET_THREAD_NUM() +
5      & INDEX
6 #endif
```


FORTRAN (cont'd) for free-form source code,

```
1 !$
```

and again,

```
1 !$  IAM = OMP_GET_THREAD_NUM() + &
2 !$&  INDEX
3 #ifdef _OPENMP
4     IAM = OMP_GET_THREAD_NUM() + &
5         INDEX
6 #endif
```


in C/C++: just use conventional preprocessing macros:

```
1 #ifdef _OPENMP
2   iam = omp_get_thread_num() + index;
3 #endif
```

"Hello, World" in OpenMP

Even though we have not covered the OpenMP directives, we can write our canonical "Hello, world" example by introducing the simplest directive to denote a parallel region:

```
1  #include <stdio.h>
2  #ifdef _OPENMP
3    #include <omp.h> /* Needed for API routines */
4  #endif
5
6  int main (int argc, char *argv[]) {
7    int th_id=0, nthreads=1;
8  #pragma omp parallel private(th_id)
9    {
10 #ifdef _OPENMP
11     th_id = omp_get_thread_num();
12 #endif
13     printf("Hello World from thread %d\n", th_id);
14 #pragma omp barrier
15     if ( th_id == 0 ) {
16 #ifdef _OPENMP
17       nthreads = omp_get_num_threads();
18 #endif
19       printf("There are %d threads\n",nthreads);
20     }
21   } /* Ends parallel region */
22   return 0;
23 }
```


OpenMP Directives

- Parallel Regions
- Work-sharing
- Data Environment
- Synchronization

OpenMP Parallel Regions

`PARALLEL/END PARALLEL` directives define parallel regions.

```
1  !$OMP PARALLEL [clause[,clause]...]
2  .
3  .
4  !$OMP END PARALLEL
```

Valid **data environment** clauses (we will come back to look at these in more detail):

- PRIVATE(list)
- SHARED(list)
- DEFAULT(PRIVATE|SHARED|NONE)
- FIRSTPRIVATE(list)
- REDUCTION({operator|intrinsic_procedure_name}:list)
- COPYIN(list)
- IF(scalar_logical_expression)
- NUM_THREADS(scalar_integer_expression)

Simple `PARALLEL` Example

```
1    integer :: myid,nthreads,npoints,ipoints,istart
2
3  !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(myid,nthreads,ipoints,istart)
4    myid = OMP_GET_THREAD_NUM()
5    nthreads = OMP_GET_NUM_THREADS()
6    ipoints = npoints/nthreads          ! size of partition by thread
7    istart = myid*ipoints+1             ! unit offset for fortran
8    if (myid.eq.nthreads-1) then
9       ipoints = npoints - istart       ! extra bits go to last thread
10   endif
11   call subdomain(x,istart,ipoints)    ! x(:) is global shared array
12 !$OMP END PARALLEL
```

- Single `PARALLEL` region in which we sub-divide the computation
- This example is more **coarse-grained**, depending on the size of the computation

OpenMP Work-Sharing

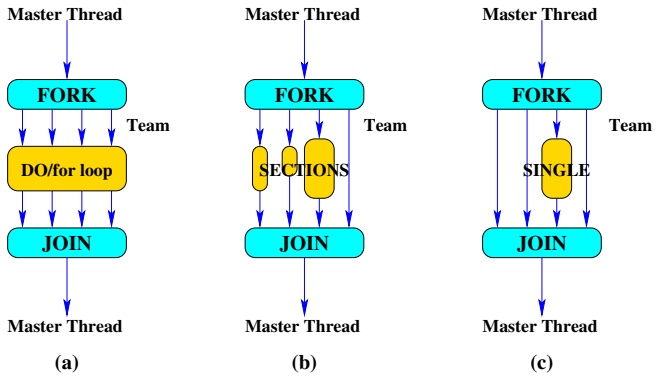
These are the real workhorses of loop-level parallelism ...

```
1 !$OMP DO [clause[,clause]...]
2 #pragma omp for [clause[,clause]...]
```

```
1 !$OMP SECTIONS [clause[,clause]...]
2 !$OMP SINGLE [clause[,clause]...]
3 !$OMP WORKSHARE
```

Let's explore these work-sharing directives one at a time ...

Work-Sharing Illustrated



Work-Sharing `DO/for` directive

```
1 !$OMP DO [clause[,clause]...]
2 #pragma omp for [clause[,clause]...]
```

- clause PRIVATE(list)
- clause FIRSTPRIVATE(list)
- clause LASTPRIVATE(list)
- clause REDUCTION({operator|intrinsic_procedure_name}:list)
- clause SCHEDULE(type[,chunk])
  control how loop iterations are mapped onto threads.
  - (static[,chunk]) chunk-sized blocks for each thread
  - (dynamic[,chunk]) threads get chunk-sized blocks until exhausted
  - (guided[,chunk]) block starts large, shrinks down to chunk size
  - (runtime) determined at runtime by env variable OMP_SCHEDULE
- clause ORDERED

Simple `DO/for` Examples

```
1  void load_arrays(int n, int m, double *A, double *B, double *C, double *D) {
2     int i;
3  #pragma omp parallel
4     {
5  #pragma omp for nowait
6        for (i=0; i<n; i++) {
7           B[i] = (A[i]-A[i-1])/2.0;
8        }
9  #pragma omp for nowait
10       for (i=0; i<m; i++) {
11          D[i] = sqrt(C[i]);
12       }
13    }
14 }
```

- Multiple (independent) loops within `PARALLEL` region
- Can use `NOWAIT` clause to avoid implicit barriers

Work-Sharing SECTIONS

```
1  !$OMP SECTIONS [clause[,clause]...]
2  [!$OMP SECTION]
3  .
4  block
5  .
6  !$OMP SECTION
7  .
8  block
9  .
10 !$OMP END SECTIONS [NOWAIT]
```

available clauses:

- PRIVATE(list)
- FIRSTPRIVATE(list)
- LASTPRIVATE(list)
- REDUCTION({operator|intrinsic_procedure_name}:list)

SECTIONS Get Assigned How?

- Implied barrier at the end of a SECTIONS directive, unless the NOWAIT clause is used
- It is not allowed to branch out of section blocks
- SECTION directives must occur within the lexical extent of a SECTIONS directive
- What if the number of threads does not match the number of sections?
  if threads $>$ sections, some threads are idled, if sections $>$ threads, implementation dependent

SECTIONS Example

```
1  !$OMP PARALLEL
2  !$OMP SECTIONS
3  !$OMP SECTION
4     CALL XAXIS()      ! xaxis(),yaxis(),zaxis() independent
5
6  !$OMP SECTION
7     CALL YAXIS()
8
9  !$OMP SECTION
10    CALL ZAXIS()
11 !$OMP END SECTIONS
12 !$OMP END PARALLEL
```

- Three subroutines executed concurrently
- Scheduling of individual SECTION blocks is implementation dependent.

Work-Sharing SINGLE

SINGLE serializes a parallel region

```
1 !$OMP SINGLE [clause[,clause]...]
2 .
3 .
4 !$OMP END SINGLE [COPYPRIVATE|NOWAIT]
```

available clauses:

- PRIVATE(list)
- FIRSTPRIVATE(list)

Example of SINGLE Directive

```
1  void single_example() {
2  #pragma omp parallel
3     {
4  #pragma omp single
5        printf("Beginning do_lots_of_work ...\n");
6
7        do_lots_of_work();
8
9  #pragma omp single
10       printf("Finished do_lots_of_work.\n");
11
12 #pragma omp single nowait
13       printf("Beginning do_lots_more_work ...\n");
14
15       do_lots_more_work();
16    }
17 }
```

- No guarantee which thread executes SINGLE region
- Can use a NOWAIT clause if other threads can continue without waiting at implicit barrier

WORKSHARE Directive

```
1  !$OMP WORKSHARE
2  .
3  .
4  !$OMP END WORKSHARE [NOWAIT]
```

available clauses:

- divides work in enclosed code into segments executed once by thread team members.
- units of work assigned in any manner subject to execution-once constraint.
- BARRIER is implied unless END WORKSHARE NOWAIT is used.

Restrictions on WORKSHARE:

- FORTRAN **only**
- Requires OpenMP version >= 2.0 (often seems to be exception even in 2.5)
- Enclosed block can only consist of:
  - array or scalar assignments
  - FORALL|WHERE statements/constructs
  - ATOMIC, CRITICAL, PARALLEL constructs

Example of WORKSHARE Directive

```
1  integer :: i,j
2  integer,parameter :: n=1000
3  real,dimension(n,n) :: A,B,C,D
4
5  do i=1,n
6    do j=1,n
7       a(i,j) = i*2.0
8       b(i,j) = j+2.0
9    enddo
10 enddo
11
12 !$OMP PARALLEL DEFAULT(SHARED)
13
14 !$OMP WORKSHARE
15  C = A*B
16  D = A+B
17  first = C(1,1)+D(1,1)
18  last = C(n,n)+D(n,n)
19 !$OMP END WORKSHARE
20
21 !$OMP END PARALLEL
```

Combined Parallel Work-Sharing

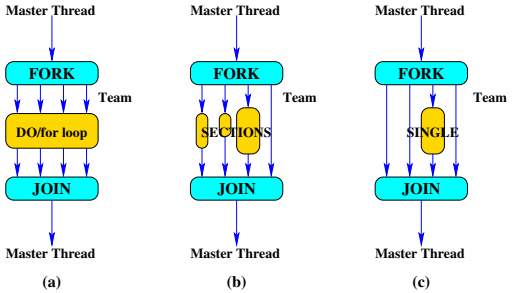
Combined Parallel region and work-sharing directives, shortcuts for regions with a single work-sharing directive,

```
1 !$OMP PARALLEL DO [clause[,clause]...]
2 #pragma omp parallel for [clause[,clause]...]
```

```
1 !$OMP PARALLEL SECTIONS [clause[,clause]...]
2 #pragma omp parallel sections [clause[,clause]...]
```

```
1 !$OMP PARALLEL WORKSHARE [clause[,clause]...]
2 #pragma omp parallel workshare [clause[,clause]...]
```


Work-Sharing Illustrated (summary)



(a) DO/for : 'data parallelism', this workhorse directive shares iterations of a loop across a thread team

(b) SECTIONS : 'functional parallelism', break work into separated discrete sections, each of which gets executed by a different thread

(c) SINGLE : serialize a section (otherwise parallel) of code

Data Environment

Constructs for controlling the data environment in parallel regions,

clause THREADPRIVATE(list) :
makes named common blocks and named variables private to each thread (Initialize with COPYIN or use DATA statements), persists between parallel regions (if thread count is the same)

clause PRIVATE(list) :
variables in list private to each member of thread team. Not initialized. Masks globals.

clause SHARED(list) :
shares variables in list among all team members (may need a FLUSH to ensure consistent copies!)

clause DEFAULT(PRIVATE|SHARED|NONE) :
C/C++ does not support DEFAULT(PRIVATE).

Data Environment (cont'd)

clause FIRSTPRIVATE(list) :
same as PRIVATE, but values initialized with value of original.

clause LASTPRIVATE(list) :
same as PRIVATE, but thread executing last iteration updates value of original object.

clause REDUCTION({operator|intrinsic_procedure_name}:list) :
performs reduction with given operator on list variables.

clause COPYIN(list) :
used in conjunction with THREADPRIVATE to initialize values in all threads.

clause COPYPRIVATE(list) :
used with END SINGLE to broadcast a value from one team member to the others.

More on the REDUCTION clause

```
1 REDUCTION (operator|intrinsic: list)
2 reduction (operator: list)
```

- A private copy of each list variable is created for each thread, the reduced value is written to the global shared variable
- listed variables must be named scalars (not arrays or structures), declared as SHARED
- Watch out for commutativity-associativity (subtraction, for example)

	FORTRAN	C/C++
intrinsic	max,min,iand,ior,ieor	
operator	$+, *, -,$ *.and.*, *.or.*, *.eqv.*, *.neqv.*	+,*,-,^,&,|,&&,||,

OpenMP 3.1 has (finally) added support for min and max in C/C++

REDUCTION Example

```
1  void ex_reduction(double *x, double *y, int n) {
2    int i,b;
3    double a;
4
5    a=0.0;
6    b=0;
7  #pragma omp parallel for private(i) shared(x,y,n) \
8                           reduction(+:a) reduction(^:b)
9     for (i=0;i<n;i++) {
10      a += x[i];
11      b ^= y[i];  /* bitwise XOR */
12    }
13 }
```


Synchronization

Directives to synchronize thread team or control thread access to code fragments,

!$OMP MASTER :
: execute section only with master thread (no implied barrier).

!$OMP CRITICAL [name]:
: restrict access to one thread at a time (otherwise block).

!$OMP BARRIER :
: synchronize all threads.

!$OMP ATOMIC :
: special case of CRITICAL, the statement following allows a specific memory location to be updated atomically (no multiple writes, can take advantage of specific hardware instructions for atomic writes).

!$OMP FLUSH [(list)]:
: ensure threads have consistent view of shared variables (else just the named list).

!$OMP ORDERED :
: execute code in same order as under sequential execution.

!$OMP SINGLE :
: block executed by only one thread (implied BARRIER and FLUSH at the end)

MASTER Example

```
1  !$OMP PARALLEL
2
3  !$OMP DO
4    do i=1,n
5       call lots_of_independent_work(i)
6    enddo
7
8  !$OMP MASTER
9    print*, 'Finished lots_of_independent_work ...'
10 !$OMP END MASTER
11
12   ... more work ...
13 !$OMP END PARALLEL
```

- Code inside MASTER construct executed only by master thread
- No implicit barrier (more efficient version of SINGLE)

Synchronization Example: ATOMIC

```
1  int main() {
2    int index[10000];
3    double x[1000],y[10000];
4    int i;
5
6    for (i=0; i<10000; i++) {
7      index[i] = i % 1000;
8      y[i] = 0.0;
9    }
10   for (i=0; i<1000; i++) {
11     x[i] = 0.0;
12   }
13   atomic_ex(x,y,index,10000);
14   return 0;
15 }
16
17 void atomic_ex(double *x, double *y, int *index, int n) {
18   int i;
19
20 #pragma omp parallel for shared(x,y,index,n)
21    for (i=0; i<n; i++) {
22 #pragma omp atomic
23       x[index[i]] += work1(i);
24      y[i] += work2(i);
25    }
26 }
```

why use ATOMIC? CRITICAL would execute serially, while ATOMIC can execute in parallel on different elements of x.

FLUSH Example

```
1  !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
2          IAM = OMP_GET_THREAD_NUM()
3          ISYNC(IAM) = 0
4  !$OMP BARRIER
5          CALL WORK()
6  !       I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
7          ISYNC(IAM) = 1
8  !$OMP FLUSH
9  !       WAIT TILL NEIGHBOR IS DONE
10         DO WHILE (ISYNC(NEIGH). EQ. 0)
11 !$OMP FLUSH(ISYNC)
12         ENDDO
13 !$OMP END PARALLEL
```

- Ensure that all threads have consistent view of memory

Example of ORDERED Clause/Construct

```
1  void work(int k) {
2  #pragma omp ordered
3     printf(" %d\n",k);
4  }
5
6  void ex_ordered(int lower, int upper, int stride) {
7    int i;
8
9  #pragma omp parallel for ordered schedule(dynamic)
10    for(i=lower, i<upper; i++) {
11       work(i);
12    }
13 }
14
15 int main() {
16   ex_ordered(0,100,5);
17   return 0;
18 }
```

- ORDERED must be within extent of PARALLEL DO, which must have ORDERED clause
- Above example prints indices in order

The task Directive

A new directive in OpenMP 3.x:

```
1  #pragma omp task [clause[[,] clause] ...]
2  {
3    /* structured block */
4  }
```

- SHARED(list)
- PRIVATE(list)
- FIRSTPRIVATE(list)
- DEFAULT(PRIVATE|SHARED|NONE)
- IF(expr)
- UNTIED (tasks not necessarily executed by parent thread)

Advantage - very good for irregular workloads (e.g., recursion, unbounded loops).

Task Synchronization

- Explicit:

```
1 #pragma omp taskwait
```

  - Encountering task waits until child tasks completed
- Implicit/Explicit:
  - tasks created by any thread of the current team guaranteed to be completed at barrier exit

More Task Details

More aspects of the `TASK` directive:

- Data scoping rules similar to `PARALLEL` regions:
  - static and global variables shared
  - automatic variables private
- lack of `DEFAULT` clause:
  - `FIRSTPRIVATE` by default

Loop `COLLAPSE`

Frequent occurrence of perfectly nested loops:

```
1 for (i=1; i<=N; i++) {
2   for (j=1; j<=M; j++) {
3     for (k=1; k<=P; k++) {
4       ...
```

In 3.0, the `COLLAPSE` directive can be used on work-sharing directives rather than attempting to nest regions (which is costly and prone to error):

```
1 #pragma omp for collapse(2)
2 for (i=1; i<=N; i++) {
3   for (j=1; j<=M; j++) {
4     for (k=1; k<=P; k++) {
5       ...
```

The compiler needs to be able to form a single loop to be parallelized - iteration space needs to be rectangular (i.e. loop indices are independent and unchanged).

Schedule Changes

There are several scheduling changes in 3.0:

- AUTO schedule - leave it to the runtime environment to decide the best schedule (very implementation dependent)
- schedule API functions (per-task control variables):

```
1 omp_set_schedule()
2 omp_get_schedule()
```

Nested Parallelism Improvements

Nested parallelism changes in 3.0:

- Per-task internal control variables (e.g., call `omp_set_num_threads` inside a parallel region to control team size at next level of parallelism
- New API routines:

```
1  /* depth of nesting */
2  omp_get_level()
3  omp_get_active_level()
4
5  /* IDs of (grand)parent */
6  omp_get_ancestor_thread_num(level)
7
8  /* team sizes of (grand)parent */
9  omp_get_team_size(level)
10
11 /* also OMP_MAX_ACTIVE_LEVELS */
12 omp_set_max_active_levels()
13 omp_get_max_active_levels()
14
15 /* also OMP_THREAD_LIMIT */
16 omp_get_thread_limit()
```


Miscellany

Environmental variables added in 3.0 (not previously mentioned):

- `OMP_STACKSIZE` - child threads' stack limit
- `OMP_WAIT_POLICY` - **active** for dedicated systems, **passive** should be good for shared systems

Major New Additions to OpenMP 4.0

OpenMP 4.0 introduced major new additions, highlights:

- **Accelerator Support**, offload to GPUs, etc.
- **User-defined Reduction Operations**, augments previous intrinsics-only
- **SIMD constructs**, portable vectorization (ILP)
- **Thread Affinity**, portable thread affinity options
- **Task Extensions**, grouping and synchronization
- **Error Handling**, new support for error handling

Note that since OpenMP 4.0 was released in 2013-07, compiler support is going to lag by some period (which may be longer for some harder to implement features).