OpenMP: An API for Writing Portable SMP Application Software

Tim Mattson
Intel Corporation
Computational Sciences Laboratory

Rudolf Eigenmann
Purdue University
School of Electrical and Computer Engineering
OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Makes it easy to create multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 15 years of SMP practice
OpenMP: Supporters*

- **Hardware vendors**
  - Intel, HP, SGI, IBM, SUN, Compaq

- **Software tools vendors**
  - KAI, PGI, PSR, APR, Absoft

- **Applications vendors**
  - ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software, and many others

*These names of these vendors were taken from the OpenMP web site (www.openmp.org). We have made no attempts to confirm OpenMP support, verify conformity to the specifications, or measure the degree of OpenMP utilization.
OpenMP: Programming Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program.
OpenMP: How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

```c
void main()
{
    double Res[1000];
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Sequential Program

```c
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Parallel Program
OpenMP: How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.

- Unintended sharing of data can lead to race conditions:
  - Race condition: when the program’s outcome changes as the threads are scheduled differently.

- To control race conditions:
  - Use synchronization to protect data conflicts.

- Synchronization is expensive so:
  - Change how data is stored to minimize the need for synchronization.
OpenMP:
Some syntax details to get us started

● Most of the constructs in OpenMP are compiler directives or pragmas.
  ◆ For C and C++, the pragmas take the form:
    
    #pragma omp construct [clause [clause]...]
  ◆ For Fortran, the directives take one of the forms:
    
    C$OMP construct [clause [clause]...]
    !$OMP construct [clause [clause]...]
    *$OMP construct [clause [clause]...]

● Since the constructs are directives, an OpenMP program can be compiled by compilers that don’t support OpenMP.
**OpenMP:**

**Structured blocks**

- Most OpenMP constructs apply to structured blocks.
  - Structured block: a block of code with one point of entry at the top and one point of exit at the bottom. The only other branches allowed are STOP statements in Fortran and exit() in C/C++.

```c
C$OMP PARALLEL
10    wrk(id) = garbage(id)
       res(id) = wrk(id)**2
       if(conv(res(id)) goto 10
C$OMP END PARALLEL

print *,id
```

```c
C$OMP PARALLEL
10    wrk(id) = garbage(id)
       res(id)=wrk(id)**2
       if(conv(res(id))goto 20
            go to 10
C$OMP END PARALLEL

if(not_DONE) goto 30
20    print *, id
```

**A structured block**

**Not A structured block**
OpenMP: Contents

- OpenMP’s constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables

- OpenMP is basically the same between Fortran and C/C++
OpenMP: Parallel Regions

- You create threads in OpenMP with the "omp parallel" pragma.
- For example, To create a 4 thread Parallel region:

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num();
    pooh(ID,A);
}
```

- Each thread redundantly executes the code within the structured block
- Each thread calls `pooh(ID)` for `ID = 0 to 3`
OpenMP: Parallel Regions

- Each thread executes the same code redundantly.

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```

A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e., a barrier)

```c
pooh(0,A) pooh(1,A) pooh(2,A) pooh(3,A)
```
OpenMP: Some subtle details (don’t worry about these at first)

- Dynamic mode (the default mode):
  - The number of threads used in a parallel region can vary from one parallel region to another.
  - Setting the number of threads only sets the maximum number of threads - you could get less.

- Static mode:
  - The number of threads is fixed and controlled by the programmer.

- OpenMP lets you nest parallel regions, but...
  - A compiler can choose to serialize the nested parallel region (i.e. use a team with only one thread).
OpenMP: Work-Sharing Constructs

- The “for” Work-Sharing construct splits up loop iterations among the threads in a team

```c
#pragma omp parallel
#pragma omp for
   for (I=0;I<N;I++){
      NEAT_STUFF(I);
   }
```

By default, there is a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.
Work Sharing Constructs
A motivating example

Sequential code

```
for(i=0; i<N; i++) { a[i] = a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart; i<iend; i++) { a[i] = a[i] + b[i]; }
}
```

OpenMP parallel region and a work-sharing for-constructor

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0; i<N; i++) { a[i] = a[i] + b[i]; }
```
The schedule clause effects how loop iterations are mapped onto threads

- `schedule(static [,chunk])`
  - Deal-out blocks of iterations of size “chunk” to each thread.

- `schedule(dynamic [,chunk])`
  - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.

- `schedule(guided [,chunk])`
  - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.

- `schedule(runtime)`
  - Schedule and chunk size taken from the OMP_SCHEDULE environment variable.
OpenMP: Work-Sharing Constructs

- The Sections work-sharing construct gives a different structured block to each thread.

```c
#pragma omp parallel
#pragma omp sections
{
    X_calculation();
#pragma omp section
    y_calculation();
#pragma omp section
    z_calculation();
}
```

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.
OpenMP: Combined Parallel Work-Sharing Constructs

- A short hand notation that combines the Parallel and work-sharing construct.

```c
#pragma omp parallel for
define I=0
for (I=0;I<N;I++){
    NEAT_STUFF(I);
}
```

- There’s also a “parallel sections” construct.
OpenMP: More details: Scope of OpenMP constructs

OpenMP constructs can span multiple source files.

static or lexical extent of parallel region

Dynamic extent of parallel region includes static extent

```
po0.f
C$OMP PARALLEL
  call whoami
C$OMP END PARALLEL

bar.f
subroutine whoami
  external omp_get_thread_num
  integer iam, omp_get_thread_num
  iam = omp_get_thread_num()
C$OMP CRITICAL
  print*, 'Hello from ', iam
C$OMP END CRITICAL
return
end
```

Orphan directives can appear outside a parallel region
Data Environment:
Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default

- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static

- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.
Data Environment:
Example storage attributes

program sort
common /input/ A(10)
inger integer index(10)
call input
C$OMP PARALLEL
  call work(index)
C$OMP END PARALLEL
print*, index(1)

subroutine work
common /input/ A(10)
real temp(10)
inger count
call work(index)
save count

A, index and count are shared by all threads.
temp is local to each thread
Data Environment: Changing storage attributes

- One can selectively change storage attributes constructs using the following clauses:
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - THREADPRIVATE

- The value of a private inside a parallel loop can be transmitted to a global value outside the loop with:
  - LASTPRIVATE

- The default status can be modified with:
  - DEFAULT (PRIVATE | SHARED | NONE)

All data clauses apply to parallel regions and worksharing constructs except “shared” which only applies to parallel regions.
Private Clause

- `private(var)` creates a local copy of `var` for each thread.
  - The value is uninitialzied
  - Private copy is *not* storage associated with the original

Regardless of initialization, `IS` is undefined at this point

```fortran
program wrong
  IS = 0
C$OMP PARALLEL DO PRIVATE(IS)
  DO J=1,1000
    IS = IS + J
  CONTINUE
  print *, IS
```
Firstprivate Clause

- Firstprivate is a special case of private.
  - Initializes each private copy with the corresponding value from the master thread.

```c
program almost_right
IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
DO J=1,1000
   IS = IS + J
1000 CONTINUE
print *, IS
```

Each thread gets its own IS with an initial value of 0

Regardless of initialization, IS is undefined at this point
Lastprivate Clause

- Lastprivate passes the value of a private from the last iteration to a global variable.

```fortran
program closer
    IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
C$OMP+ LASTPRIVATE(IS)
    DO J=1,1000
        IS = IS + J
    1000 CONTINUE
print *, IS
```

Each thread gets its own IS with an initial value of 0

IS is defined as its value at the last iteration (i.e. for J=1000)
OpenMP: Another data environment example

- Here’s an example of PRIVATE and FIRSTPRIVATE variables A, B, and C = 1

\begin{verbatim}
C$OMP PARALLEL PRIVATE(B)
C$OMP & FIRSTPRIVATE(C)
\end{verbatim}

- Inside this parallel region ...
  - “A” is shared by all threads; equals 1
  - “B” and “C” are local to each thread.
    - B’s initial value is undefined
    - C’s initial value equals 1

- Outside this parallel region ...
  - The values of “B” and “C” are undefined.
OpenMP: Default Clause

- Note that the default storage attribute is `DEFAULT(SHARED)` (so no need to specify)
- To change default: `DEFAULT(PRIVATE)`
  - each variable in static extent of the parallel region is made private as if specified in a private clause
  - mostly saves typing
- `DEFAULT(NONE)`: no default for variables in static extent. Must list storage attribute for each variable in static extent

Only the Fortran API supports default(private).
C/C++ only has default(shared) or default(none).
OpenMP:
Default Clause Example

```
итotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
    np = omp_get_num_threads()
    each = itotal/np
........
C$OMP END PARALLEL

итotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
    np = omp_get_num_threads()
    each = itotal/np
........
C$OMP END PARALLEL
```

These two codes are equivalent
Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or by using DATA statements.
A threadprivate example

Consider two different routines called within a parallel region.

```fortran
subroutine poo
parameter (N=1000)
common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
do i=1, N
  B(i) = const* A(i)
end do
return
end
```

```fortran
subroutine bar
parameter (N=1000)
common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
do i=1, N
  A(i) = sqrt(B(i))
end do
return
end
```

Because of the threadprivate construct, each thread executing these routines has its own copy of the common block /buf/.
OpenMP: Reduction

- Another clause that effects the way variables are shared:
  - `reduction (op : list)`
- The variables in “list” must be shared in the enclosing parallel region.
- Inside a parallel or a worksharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”)
  - Pair wise “op” is updated on the local value
  - Local copies are reduced into a single global copy at the end of the construct.
OpenMP:
Reduction example

#include <omp.h>
#define NUM_THREADS 2
void main ()
{
    int i;
    double ZZ, func(), res=0.0;
    omp_set_num_threads(NUM_THREADS)
#pragma omp parallel for reduction(+:res) private(ZZ)
    for (i=0; i< 1000; i++){
        ZZ = func(i);
        res = res + ZZ;
    }
}
OpenMP: Synchronization

- OpenMP has the following constructs to support synchronization:
  - atomic
  - critical section
  - barrier
  - flush
  - ordered
  - single
  - master

We discuss this here, but it really isn’t a synchronization construct. It’s a work-sharing construct that includes synchronization.
OpenMP: Synchronization

- Only one thread at a time can enter a critical section.

```c
C$OMP PARALLEL DO PRIVATE(B)
C$OMP& SHARED(RES)
    DO 100 I=1,NITERS
    B = DOIT(I)
C$OMP CRITICAL
    CALL CONSUME (B, RES)
C$OMP END CRITICAL
100 CONTINUE
```
OpenMP: Synchronization

- **Atomic** is a special case of a critical section that can be used for certain simple statements.
- It applies only to the update of a memory location (the update of X in the following example)

```c
C$OMP PARALLEL PRIVATE(B)
    B = DOIT(I)
C$OMP ATOMIC
    X = X + B
C$OMP END PARALLEL
```
**OpenMP: Synchronization**

- **Barrier**: Each thread waits until all threads arrive.

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

- Implicit barrier at the end of a parallel region.
- Implicit barrier at the end of a for work-sharing construct.
- No implicit barrier due to `nowait`.
OpenMP: Synchronization

- The `ordered` construct enforces the sequential order for a block.

```c
#pragma omp parallel private (tmp)
#pragma omp for ordered
for (i=0;i<N;i++){
    tmp = NEAT_STUFF(i);
#pragma ordered
    res = consum(tmp);
}
```
OpenMP: Synchronization

- The **master** construct denotes a structured block that is only executed by the master thread. The other threads just skip it (no implied barriers or flushes).

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
#pragma omp master
    {
        exchange_boundaries();
    }
#pragma barrier
    do_many_other_things();
}
```
OpenMP: Synchronization

- The **single** construct denotes a block of code that is executed by only one thread.
- A barrier and a flush are implied at the end of the single block.

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp single
    {     exchange_boundaries();   }
    do_many_other_things();
}
```
The flush construct denotes a sequence point where a thread tries to create a consistent view of memory.

- All memory operations (both reads and writes) defined prior to the sequence point must complete.
- All memory operations (both reads and writes) defined after the sequence point must follow the flush.
- Variables in registers or write buffers must be updated in memory.

Arguments to flush specify which variables are flushed. No arguments specifies that all thread visible variables are flushed.

This is a confusing construct and we won’t say much about it. To learn more, consult the OpenMP specifications.
OpenMP: Implicit synchronization

- Barriers are implied on the following OpenMP constructs:
  - end parallel
  - end do (except when nowait is used)
  - end sections (except when nowait is used)
  - end critical
  - end single (except when nowait is used)

- Flush is implied on the following OpenMP constructs:
  - barrier
  - critical, end critical
  - end do
  - end parallel
  - end sections
  - end single
  - ordered, end ordered
OpenMP: Some subtle details on directive nesting

- *For, sections* and *single* directives binding to the same parallel region can’t be nested.
- Critical sections with the same name can’t be nested.
- *For, sections, and single* can not appear in the dynamic extent of *critical, ordered* or *master*.
- *Barrier* can not appear in the dynamic extent of *for, ordered, sections, single, master* or *critical*.
- *Master* can not appear in the dynamic extent of *for, sections* and *single*.
- *Ordered* are not allowed inside *critical*.
- Any directives legal inside a parallel region are also legal outside a parallel region in which case they are treated as part of a team of size one.
OpenMP: Library routines

- **Lock routines**
  - `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`

- **Runtime environment routines:**
  - **Modify/Check the number of threads**
    - `omp_set_num_threads()`, `omp_get_num_threads()`,
      `omp_get_thread_num()`, `omp_get_max_threads()`
  - **Turn on/off nesting and dynamic mode**
    - `omp_set_nested()`, `omp_set_dynamic()`, `omp_get_nested()`,
      `omp_get_dynamic()`
  - **Are we in a parallel region?**
    - `omp_in_parallel()`
  - **How many processors in the system?**
    - `omp_num_procs()`
OpenMP: Library Routines

- Protect resources with locks.

```c
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp)
{
    id = omp_get_thread_num();
    tmp = do_lots_of_work(id);
   omp_set_lock(&lck);
    printf("%d %d", id, tmp);
    omp_unset_lock(&lck);
}
```
OpenMP: Library Routines

- To fix the number of threads used in a program, first turn off dynamic mode and then set the number of threads.

```c
#include <omp.h>
void main()
{
    omp_set_dynamic(0);
    omp_set_num_threads(4);
#pragma omp parallel
    {
        int id=omp_get_thread_num();
        do_lots_of_stuff(id);
    }
}
```
**OpenMP: Environment Variables**

- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
  - `OMP_SCHEDULE “schedule[, chunk_size]”`
- Set the default number of threads to use.
  - `OMP_NUM_THREADS int_literal`
- Can the program use a different number of threads in each parallel region?
  - `OMP_DYNAMIC TRUE || FALSE`
- Will nested parallel regions create new teams of threads, or will they be serialized?
  - `OMP_NESTED TRUE || FALSE`
SMP Programming errors

- Shared memory parallel programming is a mixed bag:
  - It saves the programmer from having to map data onto multiple processors. In this sense, it's much easier.
  - It opens up a range of new errors coming from unanticipated shared resource conflicts.
2 major SMP errors

- **Race Conditions**
  - The outcome of a program depends on the detailed timing of the threads in the team.

- **Deadlock**
  - Threads lock up waiting on a locked resource that will never become free.
OpenMP death-traps

- Are you using threadsafe libraries?
- I/O inside a parallel region can interleave unpredictably.
- Make sure you understand what your constructors are doing with private objects.
- Private variables can mask globals.
- Understand when shared memory is coherent. When in doubt, use FLUSH.
- NOWAIT removes implied barriers.
Navigating through the Danger Zones

- Option 1: Analyze your code to make sure every semantically permitted interleaving of the threads yields the correct results.
  - This can be prohibitively difficult due to the explosion of possible interleavings.
  - Tools like KAI’s Assure can help.
Navigating through the Danger Zones

- Option 2: Write SMP code that is portable and equivalent to the sequential form.
  - Use a safe subset of OpenMP.
  - Follow a set of “rules” for Sequential Equivalence.