OpenMP* in Action

Tim Mattson
Intel Corporation

Barbara Chapman
University of Houston

Acknowledgements:
Rudi Eigenmann of Purdue, Sanjiv Shah of Intel and others too numerous to name have contributed content for this tutorial.

The name "OpenMP" is the property of the OpenMP Architecture Review Board.

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
OpenMP Tutorial

OpenMP Overview:

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

What are Threads?

- Thread: an independent flow of control
  - Runtime entity created to execute sequence of instructions
- Threads require:
  - A program counter
  - A register state
  - An area in memory, including a call stack
  - A thread id
- A process is executed by one or more threads that share:
  - Address space
  - Attributes such as UserID, open files, working directory, etc.
A Shared Memory Architecture

Shared memory

- proc1
- proc2
- proc3
- procN
- cache1
- cache2
- cache3
- cacheN

How Can We Exploit Threads?

- A thread programming model must provide (at least) the means to:
  - Create and destroy threads
  - Distribute the computation among threads
  - Coordinate actions of threads on shared data
  - (usually) specify which data is shared and which is private to a thread
How Does OpenMP Enable Us to Exploit Threads?

- OpenMP provides thread programming model at a “high level”.
  - The user does not need to specify all the details
    - Especially with respect to the assignment of work to threads
    - Creation of threads
- User makes strategic decisions
- Compiler figures out details
- Alternatives:
  - MPI
  - POSIX thread library is lower level
  - Automatic parallelization is even higher level (user does nothing)
    - But usually successful on simple codes only

OpenMP Overview:
How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes 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 accessed to minimize the need for synchronization.
OpenMP Tutorial

Agenda

- Parallel Computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation

OpenMP:
Some syntax details to get us started

- Most of the constructs in OpenMP are compiler directives.
  - For C and C++, the directives are pragmas with the form:
    #pragma omp construct [clause [clause]…]
  - For Fortran, the directives are comments and take one of the forms:
    - Fixed form
      *$OMP construct [clause [clause]…]
      CSOMP construct [clause [clause]…]
    - Free form (but works for fixed form too)
      !$OMP construct [clause [clause]…]
- Include file and the OpenMP lib module
  #include <omp.h>
  use omp_lib
OpenMP Tutorial

OpenMP:
Structured blocks (C/C++)

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

```c
#pragma omp parallel
{   int id = omp_get_thread_num();
    more:  res[id] = do_big_job(id);
    if(!conv(res[id]) goto more;
}
printf(“ All done \n”);
```

```c
#pragma omp parallel
{   int id = omp_get_thread_num();
    more:  res[id] = do_big_job(id);
    if(!conv(res[id]) goto more;
}
printf(“ All done \n”);
```

A structured block

Not A structured block

OpenMP:
Structured Block Boundaries

- In C/C++: a block is a single statement or a group of statements between brackets {}.

```c
#pragma omp parallel
{   id = omp_thread_num();
    res(id) = lots_of_work(id);
}
```

```c
#pragma omp for
for(I=0;I<N;I++)
{   res[I] = big_calc(I);
    A[I] = B[I] + res[I];
}
```

In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```fortran
C$OMP PARALLEL
10   wrk(id) = garbage(id)
    res(id) = wrk(id)**2
    if(.not.conv(res(id)) goto 10
C$OMP END PARALLEL
```

```fortran
C$OMP PARALLEL DO
do I=1,N
    res(I)=bigComp(I)
end do
C$OMP END PARALLEL DO
```
The OpenMP* API

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_get_thread_num();
    pooh(ID,A);
}
```

- Each thread calls `pooh(ID,A)` for `ID = 0` to `3`
**The OpenMP* API**

### Parallel Regions

- Each thread executes the same code redundantly.

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

* The name “OpenMP” is the property of the OpenMP Architecture Review Board

### Exercise:

**A multi-threaded “Hello world” program**

- Write a multithreaded program where each thread prints “hello world”.

```c
void main()
{
    int ID = 0;
    printf("hello(%d) ", ID);
    printf("world(%d) \n", ID);
}
```
Exercise:
A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```c
#include "omp.h"
void main()
{
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) 
", ID);
}
}
```

Sample Output:
```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
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.

```c
#pragma omp for nowait
```

"nowait" is useful between two consecutive, independent omp for loops.

Work Sharing Constructs
A motivating example

Sequential code

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

OpenMP parallel region

```c
#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-construct

```c
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0; i<N; i++)   { a[i] = a[i] + b[i];}
```
OpenMP Tutorial

### OpenMP For/Do construct: The schedule clause

- The schedule clause affects 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_SCHEDULER environment variable.

---

### The OpenMP API

#### The schedule clause

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
</tbody>
</table>

- Least work at runtime: scheduling done at compile-time
- Most work at runtime: complex scheduling logic used at run-time
OpenMP: Work-Sharing Constructs

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

```c
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    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: Work-Sharing Constructs

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

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

- The **single** construct denotes a block of code that is executed by only one thread.
- A barrier is 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 OpenMP* API

**Combined parallel/work-share**

- **OpenMP** shortcut: Put the “parallel” and the work-share on the same line

```c
double res[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

- There’s also a “parallel sections” construct.
OpenMP Tutorial

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation

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 Sharing Examples

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

subroutine work (index)
  common /input/ A(10)
  integer index(*)
  real temp(10)
  integer count
  save count
  ..........
end subroutine work

A, index, count

temp temp temp temp temp temp temp temp
A, index, 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 uninitialized
  - Private copy is not storage-associated with the original
  - The original is undefined at the end

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

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

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

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

Firstprivate Clause

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

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

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.

    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 sequential” iteration (i.e. for J=1000)

OpenMP: A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

    variables A,B, and C = 1
    C$OMP PARALLEL PRIVATE(B)
    C$OMP& FIRSTPRIVATE(C)

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and after the parallel region?

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: Reduction

- Combines an accumulation operation across threads:
  \[ \text{reduction (op : list)} \]
- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Compiler finds standard reduction expressions containing “op” and uses them to update the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

OpenMP: Reduction example

- Remember the code we used to demo private, firstprivate and lastprivate.

```plaintext
program closer
IS = 0
DO J=1,1000
   IS = IS + J
1000 CONTINUE
print *, IS
```

- Here is the correct way to parallelize this code.

```plaintext
program closer
IS = 0
#pragma omp parallel for reduction(+:IS)
DO J=1,1000
   IS = IS + J
1000 CONTINUE
print *, IS
```
OpenMP Tutorial

OpenMP: Reduction operands/initial-values

- A range of associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial value</th>
<th>Operand</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>iand</td>
<td>All bits on</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>ior</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>ieor</td>
<td>0</td>
</tr>
<tr>
<td>.AND.</td>
<td>.true.</td>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td>.OR.</td>
<td>.false.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NEQV.</td>
<td>.false.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.EQV.</td>
<td>.true.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN*</td>
<td>Largest pos. number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX*</td>
<td>Most neg. number</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Min and Max are not available in C/C++

Default Clause

- Note that the default storage attribute is **DEFAULT(SHARED)** (so no need to use it)
- 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).
Default Clause Example

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

Are these two codes equivalent?

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

yes

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
- The runtime library and environment variables
- Recapitulation
OpenMP: Synchronization

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

The OpenMP* API

Synchronization – critical (in C/C++)

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

```c
float res;
#pragma omp parallel
{
  float B; int i;
  #pragma omp for
  for(i=0;i<niters;i++){
    B = big_job(i);
    #pragma omp critical
    consume(B, RES);
  }
}
```

Threads wait their turn – only one at a time calls consume()
**OpenMP: Synchronization**

- **Atomic** provides mutual exclusion execution but only applies to the update of a memory location (the update of X in the following example)

```c
C$OMP PARALLEL PRIVATE(B)
    B = DOIT(I)
    tmp = big_ugly();
C$OMP ATOMIC
    X = X + temp
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 due to `nowait`
**OpenMP: Synchronization**

- The *ordered* region executes in the sequential order.

```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: Implicit synchronization**

- Barriers are implied on the following OpenMP constructs:

```c
end parallel
end do  (except when nowait is used)
end sections (except when nowait is used)
end single (except when nowait is used)
```
OpenMP Tutorial

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
- The runtime library and environment variables
- Recapitulation

OpenMP: Library routines:

- 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()`
  - Are we in a parallel region?
    - `omp_in_parallel()`
  - How many processors in the system?
    - `omp_num_procs()`

...plus several less commonly used routines.
OpenMP: Library Routines

- To use a known, fixed number of threads used in a program,
  (1) tell the system that you don’t want dynamic adjustment of
  the number of threads, (2) set the number threads, then (3)
  save the number you got.

```c
#include <omp.h>
void main()
{
  int num_threads;
  omp_set_dynamic( 0 );
  omp_set_num_threads( omp_num_procs() );

  #pragma omp parallel
  {
    int id=omp_get_thread_num();
    #pragma omp single
    num_threads = omp_get_num_threads();
    do_lots_of_stuff(id);
  }
}
```

- Disable dynamic adjustment of the
  number of threads.
- Request as many threads as
  you have processors.
- Protect this op since Memory
  stores are not atomic
- Even in this case, the system may give you fewer threads
  than requested. If the precise # of threads matters, test
  for it and respond accordingly.

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

... Plus several less commonly used environment variables.
OpenMP Tutorial

Agenda
- Parallel Computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
- recapitulation

Let’s pause for a quick recap by example: Numerical Integration

Mathematically, we know that:
\[ \int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi \]

We can approximate the integral as a sum of rectangles:
\[ \sum_{i=0}^{N} F(x_i) \Delta x \approx \pi \]

Where each rectangle has width \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
PI Program: an example

```c
static long num_steps = 100000;
double step;
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

    for (i=0;i<= num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

OpenMP recap:

Parallel Region

```c
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    int i, id, nthreads; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private (i, id, x)
    {
        id = omp_get_thread_num();
        #pragma omp single
        nthreads = omp_get_num_threads();
        for (i=id, sum[id]=0.0;i< num_steps; i=i+nthreads){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```
OpenMP recap:
Synchronization (critical region)

#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2

void main ()
{
    int i, id, nthreads;    double x, pi, sum;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);

#pragma omp parallel private (i, id, x, sum)
{
    id = omp_get_thread_num();
    #pragma omp single
    nthreads = omp_get_num_threads();
    for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    #pragma omp critical
    pi += sum * step;
}
}

OpenMP recap:
Parallel for with a reduction

#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2

void main ()
{
    int i;    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);

#pragma omp parallel for private(x) reduction(+:sum)
    for (i=0;i<= num_steps; i++)
    {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
OpenMP Tutorial

OpenMP recap:
Use environment variables to set number of threads

```
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
    for (i=0;i<= num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

In practice, you set number of threads by setting the environment variable, `OMP_NUM_THREADS`