OpenMP

Amit Majumdar

Scientific Computing Applications Group
San Diego Supercomputer Center

Dept of Radiation Medicine and Applied Sciences
UCSD
Outline of Talk

- OpenMP Overview
- OpenMP Directives
- Data Scope
- Synchronization Constructs
- Run-time Library Routines
- Environment Variables
- Hands-on Exercises
Theme of the talk

- Not all OpenMP directives, clauses are mentioned and/or explained (intentionally)
- Idea is to give you an overall exposure to OpenMP and not “teach” everything about OpenMP
- I don’t remember all the directives, and all the clauses and cases where they don’t work etc…I look them up from openmp specification
- Hope is that you will have a general idea about OpenMP (if you are new to it) and will be able to think how to apply OpenMP to your serial code
OpenMP Tutorial Resources

- [http://www.openmp.org](http://www.openmp.org)
- [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)
  - Blaise Barney, Livermore Computing, LLNL
  - I have used material from the tutorial
- [http://iwomp-2012.caspur.it/program/tutorials](http://iwomp-2012.caspur.it/program/tutorials)
  - Ruud van der Pass, Senior Staff Engr, Oracle
  - I have used material from the tutorial
  - Ruud’s blogs:
      - Parallel programming using MPI, OpenMP
“Using OpenMP”
Portable Shared Memory Parallel Programming

Chapman, Jost, van der Pas

MIT Press, 2008


List price: 35 $US
Uniform Memory Access

- Also called "SMP" (Symmetric Multi Processor)
- Memory Access time is Uniform for all CPUs
- CPU can be multicore
- Interconnect is "cc":
  - Bus
  - Crossbar
- No fragmentation - Memory and I/O are shared resources

**Pro**
- Easy to use and to administer
- Efficient use of resources

**Con**
- Said to be expensive
- Said to be non-scalable
Non Uniform Memory Access

- Also called "Distributed Memory" or NORMA (No Remote Memory Access)
- Memory Access time is Non-Uniform
- Hence the name "NUMA"
- Interconnect is not "cc":
  - Ethernet, Infiniband, etc, ....
- Runs 'N' copies of the OS
- Memory and I/O are distributed resources

**Pro**
- Said to be cheap
- Said to be scalable

**Con**
- Difficult to use and administer
- In-efficient use of resources
The Hybrid Architecture

- Second-level interconnect is not cache coherent
  - Ethernet, Infiniband, etc, ....
- Hybrid Architecture with all Pros and Cons:
  - UMA within one SMP/Multicore node
  - NUMA across nodes
Two-level interconnect:

- UMA/SMP within one system
- NUMA between the systems

Both interconnects support cache coherence i.e. the system is fully cache coherent

- Has all the advantages ('look and feel') of an SMP
- Downside is the Non-Uniform Memory Access time
How to program a parallel computer

- There are numerous parallel programming models
- The ones most well-known are:
  - **Distributed Memory**
    - Sockets *(standardized, but low level)*
    - PVM – Parallel Virtual Machine
    - MPI – Message Passing Interface *(de-facto std)*
  - **Shared Memory**
    - Posix Threads *(standardized, but low level)*
    - Automatic Parallelization *(compiler does it for you)*
    - OpenMP *(de-facto standard)*
What is OpenMP

• An API to explicitly direct multi-threaded, shared memory parallelism

• Three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

• History: In the early 90s vendors of shared-memory machines supplied similar directive-based extensions
OpenMP: Programming Model

- **Fork/Join parallelism**
  - Master thread spawns threads as needed
  - Master thread always has thread ID 0
  - Parallelism is added incrementally i.e. the sequential program evolves into a parallel program
  - Explicit (not automatic) parallelism allows programmer full control over parallelism

![Diagram of Master thread and Parallel regions with multiple threads]
What is a thread

- Loosely said, a thread consists of a series of instructions with its own program counter ("PC") and state
- A parallel program executes threads in parallel
- These threads are then scheduled onto cores
The OpenMP Memory Model

- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s)
- Data transfer is through shared memory and is 100% transparent to the application
Data in OpenMP

In a shared memory parallel program variables have a "label" attached to them:

✓ Labelled "Private" \(\Rightarrow\) Visible to one thread only
  ✓ Change made in local data, is not seen by others
✓ Example - Local variables in a function that is executed in parallel
✓ Labelled "Shared" \(\Rightarrow\) Visible to all threads
  ✓ Change made in global data, is seen by all others
✓ Example - Global data
OpenMP Programming

- Most OpenMP parallelism is **Compiler Directive based**

- Programming is specified through the use compiler directives which are embedded in C/C++ or Fortran source code

- Compile with appropriate compiler option:
  - Intel compilers: `-openmp`
  - PGI compilers: `-mp`
  - IBM compilers: `-qsmp=omp`

- (OpenMP directives ignored otherwise)
Compiler Directives

- Compiler directives appear as comments in source code and are ignored unless OpenMP compiler options used.
- Compiler directives are used for:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronizing of work among threads

<table>
<thead>
<tr>
<th>Fortran</th>
<th>!$OMP PARALLEL DEFAULT(SHARED) PRIVATE (i, J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>#pragma omp parallel default(shared) private (i,j)</td>
</tr>
</tbody>
</table>
Run-time Library Routines

• Ever-growing number of run-time library routines
  – Setting and querying the number of threads
  – Querying a thread’s ID
  – Querying wall clock time and resolution
  – Setting, initializing and terminating locks

<table>
<thead>
<tr>
<th>Fortran</th>
<th>INTEGER FUNCTION OMP_GET_NUM_THREADS()</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>#include &lt;omp.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int omp_get_num_threads(void)</td>
</tr>
</tbody>
</table>
Environment Variables

• Environment variables used for controlling the execution of parallel codes during runtime such as
  – Setting the number of threads
  – Specifying how loop interactions are divided
  – Binding threads to processors
  – Setting thread stack size

• Environment setting is done in normal way

  csh/tcsh: `setenv OMP_NUM_THREADS 8`
  sh/bash: `export OMP_NUM_THREADS=8`
Outline of Talk

- OpenMP Overview
- OpenMP Directives
- Data Scope
- Synchronization Constructs
- Run-time Library Routines
- Environment Variables
- Hands-on Exercises
parallel Region Construct

• A Parallel region is a block of code that will be executed by multiple threads

Fortran :

```fortran
 !$OMP parallel [clause.....]
 structured block of code
 !$OMP end parallel
```

C/C++ :

```c
#pragma omp parallel [clause....]
 structured block of code
```
Fortran - Parallel Region Example

PROGRAM HELLO

    INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
    +    OMP_GET_THREAD_NUM

C Fork a team of threads with each thread having a private TID variable
!$OMP PARALLEL PRIVATE(TID)

C Obtain and print thread id
TID = OMP_GET_THREAD_NUM()
PRINT *, 'Hello World from thread = ', TID

C Only master thread does this
IF (TID .EQ. 0) THEN
    NTHREADS = OMP_GET_NUM_THREADS()
    PRINT *, 'Number of threads = ', NTHREADS
END IF

C All threads join master thread and disband
!$OMP END PARALLEL

END
include <omp.h>

main () {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
{

/* Obtain and print thread id */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);

/* Only master thread does this */
if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}

} /* All threads join master thread and terminate */
}
parallel Region Example

$ pgcc -mp omp_hello.c
$ export OMP_NUM_THREADS=1
$ ./a.out
Hello World from thread = 0
Number of threads = 1
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 2
Hello World from thread = 3
Hello World from thread = 1
Work Sharing Constructs

- Construct divides the execution of the enclosed code region among the members of the team that encounter it.

- No implied barrier upon entry to a work-sharing construct.

- There is implied barrier at the end of a work-sharing construct.
DO/ for loop Directive

- Shares iteration of a loop across the team
- Data parallelism

If one concept slide you had to remember from the whole talk this is it
SECTIONS Directive

- Breaks work into separate, discrete sections
- Each section is executed by a thread
- Functional parallelism
SINGLE Directive

• Serializes a section of a code
DO Directive

• Directive specifies that the iterations of the loop immediately following must be executed in parallel by the team
• A parallel region must be initiated before

```
$OMP DO [clause ...]
  SCHEDULE (type [,chunk])
  ORDERED
  PRIVATE (list)
  FIRSTPRIVATE (list)
  LASTPRIVATE (list)
  SHARED (list)
  REDUCTION (operator | intrinsic : list)
  COLLAPSE (n)

  do_loop

$OMP END DO  [ NOWAIT ]
```
#pragma omp for [clause ...] newline
schedule (type [,chunk])
ordered
private (list)
firstprivate (list)
lastprivate (list)
shared (list)
reduction (operator: list)
collapse (n)
nowait

for_loop
The Schedule Clause

- A **schedule** is specified with parallel **do/for** directive
- Describes how iterations of the loop are divided among the threads in the team
  - Correctness of your code should not depend on scheduling
  - Only performance should depend on scheduling option
- Syntax: **schedule**(type [, chunk])
- **type** can be:
  - static
  - dynamic
  - guided
  - runtime
  - auto
Example Of A Static Schedule

*A loop of length 16 using 4 threads*

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>no chunk *</td>
<td>1-4</td>
<td>5-8</td>
<td>9-12</td>
<td>13-16</td>
</tr>
<tr>
<td>chunk = 2</td>
<td>1-2</td>
<td>3-4</td>
<td>5-6</td>
<td>7-8</td>
</tr>
<tr>
<td></td>
<td>9-10</td>
<td>11-12</td>
<td>13-14</td>
<td>15-16</td>
</tr>
</tbody>
</table>
schedule (dynamic)

• For **dynamic** iterations are divided into chunks of size **chunk** (if chunk is not present, the size of all chunks is 1)

• At runtime, chunks are assigned to threads dynamically

• As threads become available they pick up chunks

• Useful for **load balance** if different chunks do different amount of work
The first chunk size is implementation dependent
Size of successive chunks decreases each time a parcel of work is given to a thread

If chunk is not specified minimum chunk size is one
Initial block:
\[
\frac{\text{\#\_of\_iterations}}{\text{\#\_of\_threads}}
\]
Subsequent blocks:
\[
\frac{\text{\#\_of\_iterations\_remaining}}{\text{\#\_of\_threads}}
\]
schedule (auto and runtime)

• auto
  – The compiler (or runtime system) decides what is best to use

• runtime
  – The schedule type is chosen at runtime based on the environment variable \texttt{omp\_schedule}
  – No chunk size allowed
PROGRAM VEC_ADD_DO

INTEGER N, CHUNKSIZE, CHUNK, I
PARAMETER (N=1000)
PARAMETER (CHUNKSIZE=100)
REAL A(N), B(N), C(N)

! Some initializations
DO I = 1, N
   A(I) = I * 1.0
   B(I) = A(I)
ENDDO
CHUNK = CHUNKSIZE

!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)

!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
   DO I = 1, N
      C(I) = A(I) + B(I)
   ENDDO
!$OMP END DO NOWAIT

!$OMP END PARALLEL

END
#include <omp.h>
define CHUNKSIZE 100ndefine N 1000

main ()
{

int i, chunk;
float a[N], b[N], c[N];

// Some initializations */
for (i=0; i < N; i++)
   a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
{

#pragma omp for schedule(dynamic,chunk) nowait
for (i=0; i < N; i++)
   c[i] = a[i] + b[i];

} /* end of parallel section */
}
Combined Parallel Work Sharing Constructs (convenience)

Fortran - PARALLEL DO Directive Example

```fortran
PROGRAM VECTOR_ADD

INTEGER N, I, CHUNKSIZE, CHUNK
PARAMETER (N=1000)
PARAMETER (CHUNKSIZE=100)
REAL A(N), B(N), C(N)

! Some initializations
DO I = 1, N
   A(I) = I * 1.0
   B(I) = A(I)
ENDDO

CHUNK = CHUNKSIZE

!$OMP PARALLEL DO
!$OMP& SHARED(A,B,C,CHUNK) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)

   DO I = 1, N
      C(I) = A(I) + B(I)
   ENDDO

!$OMP END PARALLEL DO

END
```

C / C++ - parallel for Directive Example

```c
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main () {

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
   a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel for \
   shared(a,b,c,chunk) private(i) \
   schedule(static,chunk)
for (i=0; i < n; i++)
   c[i] = a[i] + b[i];
}
```
SECTIONS Directive

- Non-iterative work-sharing construct
- Enclosed section(s) of code are to be divided among the threads in the team
- Each SECTION is executed once by a thread in the team
- Different sections may be executed by different threads
- It is possible for a thread to execute more than one section (e.g. more sections than threads)
Fortran:

```
!$OMP SECTIONS [clause ...]
 PRIVATE (list)
 FIRSTPRIVATE (list)
 LASTPRIVATE (list)
 REDUCTION (operator | intrinsic : list)

!$OMP SECTION

 block

!$OMP SECTION

 block

!$OMP END SECTIONS  [ NOWAIT ]
```

C/C++:

```
#pragma omp sections [clause ...] newline
 private (list)
 firstprivate (list)
 lastprivate (list)
 reduction (operator: list)
 nowait

{

 #pragma omp section  newline

  structured_block

 #pragma omp section  newline

  structured_block

}
```
Fortran - SECTIONS Directive Example

PROGRAM VEC_ADD_SECTIONS

INTEGER N, I
PARAMETER (N=1000)
REAL A(N), B(N), C(N), D(N)

! Some initializations
DO I = 1, N
   A(I) = I * 1.5
   B(I) = I + 22.35
ENDDO

!$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)

!$OMP SECTIONS

!$OMP SECTION
   DO I = 1, N
      C(I) = A(I) + B(I)
   ENDDO

!$OMP SECTION
   DO I = 1, N
      D(I) = A(I) * B(I)
   ENDDO

!$OMP END SECTIONS NOWAIT

!$OMP END PARALLEL

END
C / C++ - sections Directive Example

#include <omp.h>
define N 1000

main ()
{
    int i;
    float a[N], b[N], c[N], d[N];

    /* Some initializations */
    for (i=0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }

    #pragma omp parallel shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++)
                c[i] = a[i] + b[i];

            #pragma omp section
            for (i=0; i < N; i++)
                d[i] = a[i] * b[i];
        } /* end of sections */
    } /* end of parallel section */
}
SINGLE Directive

- Specifies that the enclosed code is executed by only one thread
- Useful when doing for e.g. I/O

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$OMP SINGLE [clause ...] PRIVATE (list) FIRSTPRIVATE (list) block</td>
<td></td>
</tr>
<tr>
<td>!$OMP END SINGLE [ NOWAIT ]</td>
<td></td>
</tr>
<tr>
<td>#pragma omp single [clause ...] newline private (list) firstprivate (list) nowait</td>
<td></td>
</tr>
<tr>
<td>structured_block</td>
<td></td>
</tr>
</tbody>
</table>
SINGLE Directive Example

Original Code

```c
.....
"read A[0..N-1];"
.....
```

#pragma omp parallel \
    shared (A)
{
    ..... 
    #pragma omp single nowait
    {"read A[0..N-1];"}
    ..... 
    #pragma omp barrier
    "use A"
}

Parallel Version
Outline of Talk

- OpenMP Overview
- OpenMP Directives
- Data Scope
- Synchronization Constructs
- Run-time Library Routines
- Environment Variables
- Hands-on Exercises
Data Scoping

- Important consideration of OpenMP programming
- OpenMP is based on shared memory programming model and most variables are shared by default
- OpenMP Data Scope Attributes are used to explicitly define how variables be scoped
  - PRIVATE
  - SHARED
  - DEFAULT
  - FIRSTPRIVATE/LASTPRIVATE
  - REDUCTION
  - COPYIN (later)
PRIVATE Clause

- A new object of the same type is declared once for each thread
- References to the original object are replaced with ref to the new one
- They are assumed to be uninitialized for each thread

<table>
<thead>
<tr>
<th>Fortran</th>
<th>PRIVATE (list)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>private (list)</td>
</tr>
</tbody>
</table>
SHAREDClause

- Declares variables in its list to be shared among all the threads
- A shared variable exists in only one memory location and all threads read or write to that address
- Programmers duty to make sure they are accessed properly

<table>
<thead>
<tr>
<th>Fortran</th>
<th>SHAREDC(list)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>shared (list)</td>
</tr>
</tbody>
</table>
DEFAULT Clause

- This allows users to specify a default scope for all variables within a parallel region
- Then use PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DEFAULT (PRIVATE</td>
<td>FIRSTPRIVATE</td>
</tr>
</tbody>
</table>
**FIRSTPRIVATE/LASTPRIVATE Clause**

- **FIRSTPRIVATE**: All variables in the list are initialized with the value the original object had before entering the parallel construct.

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRSTPRIVATE (list)</td>
<td>firstprivate (list)</td>
</tr>
</tbody>
</table>

- **LASTPRIVATE**: The thread that executes the sequentially last iteration or section updates the value of the objects in the list.

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>LASTPRIVATE (list)</td>
<td>lastprivate (list)</td>
</tr>
</tbody>
</table>
n = 2; indx = 4;
#pragma omp parallel default(none) private(i,TID)
    firstprivate(indx) shared(n,a)
{
    TID = omp_get_thread_num();
    indx = indx + n*TID;
    for(i=indx; i<indx+n; i++)
        a[i] = TID + 1;
} /*-- End of parallel region --*/
Example Lastprivate

```c
#include <omp.h>

#pragma omp parallel for lastprivate(a)
for (int i=0; i<n; i++)
{
    .......
    a = i + 1;
    .......
} // End of parallel region

b = 2 * a; // value of b is 2*n
```
REDUCTION Clause

- Performs a reduction on the variables that appear in the list
- A private copy for each list variable is created for each thread
- At the end of the reduction, the reduction variable is applied to all private copies of the shared variable and the final result is written to the global variable

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>REDUCTION (operator</td>
</tr>
<tr>
<td>C/C++</td>
<td>reduction (operator: list)</td>
</tr>
</tbody>
</table>
Fortran - REDUCTION Clause Example

PROGRAM DOT_PRODUCT

INTEGER N, CHUNKSIZE, CHUNK, I
PARAMETER (N=100)
PARAMETER (CHUNKSIZE=10)
REAL A(N), B(N), RESULT

! Some initializations
DO I = 1, N
   A(I) = I * 1.0
   B(I) = I * 2.0
ENDDO
RESULT = 0.0
CHUNK = CHUNKSIZE

 !$OMP PARALLEL DO
 !$OMP& DEFAULT(SHARED) PRIVATE(I)
 !$OMP& SCHEDULE(STATIC, CHUNK)
 !$OMP& REDUCTION(+:RESULT)

   DO I = 1, N
      RESULT = RESULT + (A(I) * B(I))
   ENDDO

 !$OMP END PARALLEL DO

PRINT *, 'Final Result= ', RESULT
END

C / C++ - reduction Clause Example

#include <omp.h>

main () {

int i, n, chunk;
float a[100], b[100], result;

/* Some initializations */

n = 100;
chunk = 10;
result = 0.0;
for (i=0; i < n; i++)
{
   a[i] = i * 1.0;
   b[i] = i * 2.0;
}

#pragma omp parallel for \
   default(shared) private(i) \
   schedule(static,chunk) \
   reduction(+:result)

   for (i=0; i < n; i++)
      result = result + (a[i] * b[i]);

printf("Final result= %f\n",result);
}

54
nowait clause

- To minimize synchronization, some directives support the optional nowait clause
  - If present, threads do not synchronize/wait at the end of that particular construct
- In C, it is one of the clauses on the pragma
- In Fortran, it is appended at the closing part of the construct

```c
#pragma omp for nowait
{
    
}
```
Outline of Talk

- OpenMP Overview
- OpenMP Directives
- Data Scope
- Synchronization Constructs
- Run-time Library Routines
- Environment Variables
- Hands-on Exercises
Data Races

In a shared memory program, care needs to be taken when updating a shared variable

Example:
Thread 0: X = 1
Thread 1: X = 2

The resulting value for “X” is undefined

This is an example of a “data race”
- Two different threads in a multi-threaded shared memory program
- Access the same (=shared) memory location
  - Asynchronously and
  - Without holding any common exclusive locks and
  - At least one of the accesses is a write/store
Example of Data Race

```c
#pragma omp parallel shared(n)
{n = omp_get_thread_num();}
```
Another example of data race

```c
#pragma omp parallel shared(x)
{x = x + 1;}
```
Data Race

• Data race means that the update of a shared variable is not well protected
• Numerical results can differ run to run
• Depending on number of threads result can appear or disappear etc.
Synchronization Constructs

- These control how the execution of one thread proceeds relative to other

**MASTER Directive**

- Specifies a region to be executed only by the master thread
- All other threads skip (no implied barrier)

```
Fortran

 !$OMP MASTER

 block

 !$OMP END MASTER

C/C++

 #pragma omp master newline

 structured_block
```
The enclosed block of code is executed in the order in which iterations would be executed sequentially:

```
#pragma omp ordered
{<code-block>}

 !$omp ordered
  <code-block>
 !$omp end ordered
```

May introduce serialization (could be expensive)
CRITICAL Directive

- Specifies a region of code that must be executed by only one thread at a time

```fortran
$OMP CRITICAL [ name ]
  block
$OMP END CRITICAL [ name ]
```

```c
#pragma omp critical [ name ] newline
  structured_block
```
If sum is a shared variable, this loop can not run in parallel by simply using a "#pragma omp for"

```c
for (i=0; i < n; i++){
    ....
    sum += a[i];
    ....
}
```

```c
#pragma omp parallel for
for (i=0; i < n; i++){
    ....
    #pragma omp critical
    {sum += a[i];}
    ....
}
```

All threads execute the update, but now only one at a time will do so.
Fortran - CRITICAL Directive Example

PROGRAM CRITICAL

INTEGER X
X = 0

!$OMP PARALLEL SHARED(X)

!$OMP CRITICAL
X = X + 1
!$OMP END CRITICAL

!$OMP END PARALLEL

END
BARRIER Directive

- Synchronizes all the threads in the team
- A thread will wait at BARRIER until all other threads join there

<table>
<thead>
<tr>
<th>Fortran</th>
<th>!$OMP BARRIER</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>#pragma omp barrier newline</td>
</tr>
</tbody>
</table>
**ATOMIC Directive**

- This directive specifies that a specific memory location must be updated atomically and not letting multiple threads attempt to write to it.
- Essentially a mini-CRITICAL section.

### Fortran

```fortran
$OMP ATOMIC
statement_expression
```

### C/C++

```c
#pragma omp atomic
statement_expression
```

---

**Atomic: only the loads and store are atomic ....**

- **Fortran**
  ```fortran
  !$OMP ATOMIC
  <statement>
  ```

- **C/C++**
  ```c
  #pragma omp atomic
  <statement>
  ```

**This is a lightweight, special form of a critical section**

```c
#pragma omp atomic
a[ indx[i] ] += b[i];
```
THREADPRIVATE Directive

- Used to make global file scope variables (C/C++) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$OMP THREADPRIVATE (/cb/, ...) cb is the name of a common block</td>
<td>#pragma omp threadprivate (list)</td>
</tr>
</tbody>
</table>
$ pgcc -mp threadprivate.c
$ ./a.out
1st Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 2:   a,b,x= 2 2 3.200000
Thread 3:   a,b,x= 3 3 4.300000
Thread 1:   a,b,x= 1 1 2.100000
******************************
***
Master thread doing serial work here
******************************
***
2nd Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 1:   a,b,x= 1 0 2.100000
Thread 2:   a,b,x= 2 0 3.200000
Thread 3:   a,b,x= 3 0 4.300000

#include <omp.h>

int a, b, i, tid;
float x;

#pragma omp threadprivate(a, x)

int main ()
{
    /* Explicitly turn off dynamic threads */
    omp_set_dynamic(0);

    printf("1st Parallel Region:\n");
    #pragma omp parallel private(b,tid)
    {
        tid = omp_get_thread_num();
        a = tid;
        b = tid;
        x = 1.1 * tid + 1.0;
        printf("Thread %d:  a,b,x= %d %d %f\n",tid,a,b,x);
    }  /* end of parallel section */

    printf("\n");
    printf("Master thread doing serial work here\n");
    printf("\n");

    printf("2nd Parallel Region:\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("Thread %d:  a,b,x= %d %d %f\n",tid,a,b,x);
    }  /* end of parallel section */
}
PROGRAM THREADPRIV

INTEGER A, B, I, TID, OMP_GET_THREAD_NUM
REAL*4 X
COMMON /C1/ A

 !$OMP THREADPRIVATE(/C1/, X)

C Explicitly turn off dynamic threads
CALL OMP_SET_DYNAMIC(.FALSE.)

PRINT *, '1st Parallel Region:'
 !$OMP PARALLEL PRIVATE(B, TID)
 TID = OMP_GET_THREAD_NUM()
 A = TID
 B = TID
 X = 1.1 * TID + 1.0
 PRINT *, 'Thread',TID,: A,B,X
 !$OMP END PARALLEL

 PRINT *, '****************************'
 PRINT *, 'Master thread doing serial work here'
 PRINT *, '****************************'

 PRINT *, '2nd Parallel Region: '
 !$OMP PARALLEL PRIVATE(TID)
 TID = OMP_GET_THREAD_NUM()
 PRINT *, 'Thread',TID,: A,B,X
 !$OMP END PARALLEL

END
COPYIN Clause

- Applies to THREADPRIVATE common blocks
- At the start of the parallel region, data of the master thread is copied to the thread private copies

Example:

```plaintext
common /cblock/velocity
common /fields/xfield, yfield, zfield

! create thread private common blocks

!$omp threadprivate (/cblock/, /fields/)

!$omp parallel &
!$omp default (private) &
!$omp copyin ( /cblock/, zfield )
```
Outline of Talk

- OpenMP Overview
- OpenMP Directives
- Data Scope
- Synchronization Constructs
- Run-time Library Routines
- Environment Variables
- Hands-on Exercises
• **OpenMP provides a set of runtime functions**
• **They all start with “omp_”**
• **These functions can be used to:**
  ✓ **Query for a specific feature**
    • **For example, the current iteration scheduling policy for loops**
  ✓ **Change the setting**
    • **For example, to change the iteration scheduling policy for loops**
• **A special category consists of the locking functions**

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Need to include file <code>&lt;omp.h&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>Add “use omp_lib” or include file “omp_lib.h”</td>
</tr>
<tr>
<td>Name</td>
<td>Functionality</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td><code>omp_set_num_threads</code></td>
<td>Set number of threads</td>
</tr>
<tr>
<td><code>omp_get_num_threads</code></td>
<td>Number of threads in team</td>
</tr>
<tr>
<td><code>omp_get_max_threads</code></td>
<td>Max num of threads for parallel region</td>
</tr>
<tr>
<td><code>omp_get_thread_num</code></td>
<td>Get thread ID</td>
</tr>
<tr>
<td><code>omp_get_num_procs</code></td>
<td>Maximum number of processors</td>
</tr>
<tr>
<td><code>omp_in_parallel</code></td>
<td>Check whether in parallel region</td>
</tr>
<tr>
<td><code>omp_set_dynamic</code></td>
<td>Activate dynamic thread adjustment</td>
</tr>
<tr>
<td></td>
<td>(but implementation is free to ignore this)</td>
</tr>
<tr>
<td><code>omp_get_dynamic</code></td>
<td>Check for dynamic thread adjustment</td>
</tr>
<tr>
<td><code>omp_set_nested</code></td>
<td>Activate nested parallelism</td>
</tr>
<tr>
<td></td>
<td>(but implementation is free to ignore this)</td>
</tr>
<tr>
<td><code>omp_get_nested</code></td>
<td>Check for nested parallelism</td>
</tr>
<tr>
<td><code>omp_get_wtime</code></td>
<td>Returns wall clock time</td>
</tr>
<tr>
<td><code>omp_get_wtick</code></td>
<td>Number of seconds between clock ticks</td>
</tr>
<tr>
<td>Name</td>
<td>Functionality</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>omp_set_schedule</td>
<td>Set schedule (if “runtime” is used)</td>
</tr>
<tr>
<td>omp_get_schedule</td>
<td>Returns the schedule in use</td>
</tr>
<tr>
<td>omp_get_thread_limit</td>
<td>Max number of threads for program</td>
</tr>
<tr>
<td>omp_set_max_active_levels</td>
<td>Set number of active parallel regions</td>
</tr>
<tr>
<td>omp_get_max_active_levels</td>
<td>Number of active parallel regions</td>
</tr>
<tr>
<td>omp_get_level</td>
<td>Number of nested parallel regions</td>
</tr>
<tr>
<td>omp_get_active_level</td>
<td>Number of nested active par. regions</td>
</tr>
<tr>
<td>omp_get_ancestor_thread_num</td>
<td>Thread id of ancestor thread</td>
</tr>
<tr>
<td>omp_get_team_size (level)</td>
<td>Size of the thread team at this level</td>
</tr>
</tbody>
</table>
Outline of Talk

- OpenMP Overview
- OpenMP Directives
- Data Scope
- Synchronization Constructs
- Run-time Library Routines
- Environment Variables
- Hands-on Exercises
<table>
<thead>
<tr>
<th>OpenMP Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_NUM_THREADS</td>
</tr>
<tr>
<td>OMP_SCHEDULE “schedule,[chunk]”</td>
</tr>
<tr>
<td>OMP_DYNAMIC {TRUE</td>
</tr>
<tr>
<td>OMP_NESTED {TRUE</td>
</tr>
<tr>
<td>OMP_STACKIZE “size [B</td>
</tr>
<tr>
<td>OMP_WAIT_POLICY [ACTIVE</td>
</tr>
<tr>
<td>OMP_MAX_ACTIVE_LEVELS</td>
</tr>
<tr>
<td>OMP_THREAD_LIMIT</td>
</tr>
</tbody>
</table>
Outline of Talk

- OpenMP Overview
- OpenMP Directives
- Data Scope
- Synchronization Constructs
- Run-time Library Routines
- Environment Variables
- Hands-on Exercises
- Will be done on the trestles.sdsc.edu machine
- Log in and copy openmp.tar from /home/majumdar/OpenMP directory (command: `cp /home/majumdar/openmp.tar .`)
- **Untar it** (command: `tar xvf openmp.tar`)
- See the four codes:
  - `mm.c` `omp_hello.c` `omp_reduction.c` `omp_workshare.c`
  - `mm.f` `omp_hello.f` `omp_reduction.f` `omp_workshare.f`
- Compile as follows:
  - `pgcc –mp omp_hello.c`
  - `pgf90 –mp omp_hello.f`
- Modify the Trestles_batch_script
- Submit job: `qsub Trestles_batch_script`
- Check results:
  - matrix multiply (mm) and reduction (omp_reduction) codes are for timing results
  - omp_hello and omp_workshare are for functionalities
```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {  
    int nthreads, tid;  
    /* Fork a team of threads giving them their own copies of variables */  
    #pragma omp parallel private(nthreads, tid) {  
        /* Obtain thread number */  
        tid = omp_get_thread_num();  
        printf("Hello World from thread = %d\n", tid);  
        /* Only master thread does this */  
        if (tid == 0) {  
            nthreads = omp_get_num_threads();  
            printf("Number of threads = %d\n", nthreads);  
        }  
        } /* All threads join master thread and disband */  
    }
```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N_v 90000000

int main (int argc, char *argv[])
{
    int i, tid, nthreads;
    double a[N_v], b[N_v], c[N_v], d[N_v], sum1, sum2;
    double tstart, tstop;
    tstart = omp_get_wtime();
    /**< Spawn a parallel region explicitly scoping all variables */
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
        }
    }
    tstop = omp_get_wtime();
    printf("%f\n", tstop - tstart);
}
/ * Some initializations */
sum1 = sum2 = 0.0;

#pragma omp parallel for reduction(+:sum1)
for (i=0; i < N_v; i++) {
    a[i] = i*0.0001/5. + pow(0.75,8) + pow(cos(3.14),5);
    c[i] = pow(0.5,9) + sin(3.14) + pow(tan(3.14),5);
    sum1 = sum1 + (a[i] * c[i]);
}

#pragma omp parallel for reduction(+:sum2)
for (i=0; i < N_v; i++) {
    b[i] = i*0.0001/5.+ pow(0.75,8) + pow(cos(3.14),5);
    d[i] = pow(0.5,9) + sin(3.14) + pow(tan(3.14),5);
    sum2 = sum2 + (b[i] * d[i]);
}

tstop = omp_get_wtime();

printf("   Result (should be zero) = %f
",sum1 - sum2);

printf("wtime =%.6g on %d threads
",tstop - tstart, nthreads);
#include <omp.h>

#include <stdio.h>
#include <stdlib.h>

#define NRA 2000                 /* number of rows in matrix A */
#define NCA 2000                 /* number of columns in matrix A */
#define NCB 2000                  /* number of columns in matrix B */

int main (int argc, char *argv[])
{
    int     tid, nthreads, i, j, k, chunk;
    double  a[NRA][NCA],           /* matrix A to be multiplied */
            b[NCA][NCB],           /* matrix B to be multiplied */
            c[NRA][NCB],           /* result matrix C */
            tstart,               
            tstop;

    tstart = omp_get_wtime();
    chunk = 10;                  /* set loop iteration chunk size */
/** Spawn a parallel region explicitly scoping all variables ***/

```c
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
{
    tid =omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
    }

    /*** Initialize matrices ***/
    #pragma omp for schedule (static, chunk)
    for (i=0; i<NRA; i++)
        for (j=0; j<NCA; j++)
            a[i][j]= i+j;

    #pragma omp for schedule (static, chunk)
    for (i=0; i<NCA; i++)
        for (j=0; j<NCB; j++)
            b[i][j]= i*j;

    #pragma omp for schedule (static, chunk)
    for (i=0; i<NRA; i++)
        for (j=0; j<NCB; j++)
            c[i][j]= 0;
```
/*** Do matrix multiply sharing iterations on outer loop ***/

#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
{
    for(j=0; j<NCB; j++)
    for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];
}

/*** End of parallel region ***/

tstop = omp_get_wtime();
    printf("wtime =%.6g on %d threads
",tstop - tstart, nthreads);

printf("*********************************************************************************
");
printf ("Done.
");
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define N 5

int main (int argc, char *argv[])
{
    int i, nthreads, tid;
    float a[N], b[N], c[N], d[N];

    /* Some initializations */
    for (i=0; i<N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
        c[i] = d[i] = 0.0;
    }
}
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)  
   
   tid = omp_get_thread_num();  
   if (tid == 0)  
       {  
           nthreads = omp_get_num_threads();  
           printf("Number of threads = %d\n", nthreads);  
       }  
   printf("Thread %d starting...\n",tid);  
   
   #pragma omp sections nowait  
       {  
           #pragma omp section  
           {  
               printf("Thread %d doing section 1\n",tid);  
               for (i=0; i<N; i++)  
               {  
                   c[i] = a[i] + b[i];  
                   printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);  
               }  
           }  
       }
#pragma omp section
{
    printf("Thread %d doing section 2\n",tid);
    for (i=0; i<N; i++)
    {
        d[i] = a[i] * b[i];
        printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
    }
}

} /* end of sections */

printf("Thread %d done.\n",tid);

} /* end of parallel section */
The End
OpenMP Fortran Directives

• Fortran
  !$OMP directive
   [ structured block of code}
  !$OMP end directive

• Several Fortran directives come in pairs as shown above

• Fixed source format
  – All fixed source format rules apply
  – !$OMP, C$OMP, *$OMP are accepted and must start in column 1
  – Continuation lines must have a non-space/zero in column 6

• Free source format
  – !$OMP is the only accepted format
  – All free source format rules apply
  – Continuation line must have an ampersand as the last non-blank character
    The following line must begin with a sentinel and then the continuation directives