Prof. Thomas Sterling
Pervasive Technology Institute
School of Informatics & Computing
Indiana University

Dr. Steven R. Brandt
Center for Computation & Technology
Louisiana State University

Basics of Supercomputing

Programming in OpenMP
Topics

• Introduction to OpenMP
• Runtime library and environment variables
• Data and work sharing directives
• Synchronization
• Reduction
Introduction

• OpenMP is:
  – An API (Application Programming Interface)
  – NOT a programming language
  – A set of compiler directives that help the application developer to parallelize their workload.
  – Includes environment variables and the library routines
# Components of OpenMP

## Directives
- Parallel regions
- Work sharing
- Synchronization
- Data scope attributes:  
  - private  
  - firstprivate  
  - last private  
  - shared  
  - reduction
- Orphaning

## Runtime library routines
- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested Parallelism
- Timers
- API for locking

## Environment variables
- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested Parallelism
OpenMP Architecture

Inspired by OpenMp.org introductory slides
Topics

- Introduction to OpenMP
- Runtime library and environment variables
- Data and work sharing directives
- Synchronization
- Reduction
Runtime Library Routines

- Runtime library routines help manage parallel programs
- Many runtime library routines have corresponding environment variables that can be controlled by the users
- Runtime libraries can be accessed by including `omp.h` in applications that use OpenMP: `#include <omp.h>`
- For example for calls like:
  - `omp_get_num_threads()`, (by which an openMP program determines the number of threads available for execution) can be controlled using an environment variable set at the command-line of a shell (`$OMP_NUM_THREADS`)
- Some of the activities that the OpenMP libraries help manage are:
  - Determining the number of threads/processors
  - Scheduling policies to be used
  - General purpose locking and portable wall clock timing routines
### OpenMP: Runtime Library

#### Function: `omp_get_num_threads()`

<table>
<thead>
<tr>
<th>C/ C++</th>
<th><code>int omp_get_num_threads(void);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td><code>integer function omp_get_num_threads()</code></td>
</tr>
</tbody>
</table>

**Description:**
Returns the total number of threads currently in the group executing the parallel block from where it is called.

#### Function: `omp_get_thread_num()`

<table>
<thead>
<tr>
<th>C/ C++</th>
<th><code>int omp_get_thread_num(void);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td><code>integer function omp_get_thread_num()</code></td>
</tr>
</tbody>
</table>

**Description:**
For the master thread, this function returns zero. For the child nodes the call returns an integer between 1 and `omp_get_num_threads()-1` inclusive.
OpenMP Environment Variables

- OpenMP provides 4 main environment variables for controlling execution of parallel codes:
  - **OMP_NUM_THREADS** – controls the parallelism of the OpenMP application
  - **OMP_DYNAMIC** – enables dynamic adjustment of number of threads for execution of parallel regions
  - **OMP_SCHEDULE** – controls the load distribution in loops such as *do, for*
  - **OMP_NESTED** – enables nested parallelism in OpenMP applications
## OpenMP Environment Variables

<table>
<thead>
<tr>
<th>Environment Variable:</th>
<th>OMP_NUM_THREADS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage:</strong></td>
<td>OMP_NUM_THREADS ( n )</td>
</tr>
<tr>
<td>bash/sh/ksh:</td>
<td><code>export OMP_NUM_THREADS=8</code></td>
</tr>
<tr>
<td>csh/tcsh</td>
<td><code>setenv OMP_NUM_THREADS 8</code></td>
</tr>
</tbody>
</table>

**Description:**
Sets the number of threads to be used by the OpenMP program during execution.

<table>
<thead>
<tr>
<th>Environment Variable:</th>
<th>OMP_DYNAMIC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage:</strong></td>
<td>OMP_DYNAMIC {TRUE|FALSE}</td>
</tr>
<tr>
<td>bash/sh/ksh:</td>
<td><code>export OMP_DYNAMIC=TRUE</code></td>
</tr>
<tr>
<td>csh/tcsh</td>
<td><code>setenv OMP_DYNAMIC “TRUE”</code></td>
</tr>
</tbody>
</table>

**Description:**
When this environment variable is set to TRUE the maximum number of threads available for use by the OpenMP program is \( n \) ($OMP_NUM_THREADS$).
### OpenMP Environment Variables

<table>
<thead>
<tr>
<th>Environment Variable:</th>
<th>OMP_SCHEDULE</th>
</tr>
</thead>
</table>

**Usage:**

<table>
<thead>
<tr>
<th>bash/sh/ksh:</th>
<th>export OMP_SCHEDULE static,N/P</th>
</tr>
</thead>
<tbody>
<tr>
<td>csh/tcsh</td>
<td>setenv OMP_SCHEDULE=&quot;GUIDED,4&quot;</td>
</tr>
</tbody>
</table>

**Description:**

Only applies to *for* and *parallel for* directives. This environment variable sets the schedule type and chunk size for all such loops. The chunk size can be provided as an integer number, the default being 1.

<table>
<thead>
<tr>
<th>Environment Variable:</th>
<th>OMP_NESTED</th>
</tr>
</thead>
</table>

**Usage:**

<table>
<thead>
<tr>
<th>bash/sh/ksh:</th>
<th>export OMP_NESTED FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>csh/tcsh</td>
<td>setenv OMP_NESTED=&quot;FALSE&quot;</td>
</tr>
</tbody>
</table>

**Description:**

Setting this environment variable to **TRUE** enables multi-threaded execution of inner parallel regions in nested parallel regions.
OpenMP: Basic Constructs

OpenMP Execution Model (FORK/JOIN):

Sequential Part (master thread)
Parallel Region (FORK: group of threads)
Sequential Part (JOIN: master thread)
Parallel Region (FORK: group of threads)
Sequential Part (JOIN: master thread)

C / C++ :

```c
#pragma omp parallel
{
    parallel block
} /* omp end parallel */
```

To invoke library routines in C/C++ add

```c
#include <omp.h>
```

near the top of your code
```c
#include <omp.h>

main ()
{
    int nthreads, tid;
    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
}
```

Non shared copies of data for each thread

OpenMP directive to indicate START segment to be parallelized

Code segment that will be executed in parallel

OpenMP directive to indicate END segment to be parallelized
OpenMP Execution

• On encountering the C construct `#pragma omp parallel`, \( n-1 \) extra threads are created

• \texttt{omp_get_thread_num()} returns a unique identifier for each thread that can be utilized. The value returned by this call is between 0 and \((\texttt{OMP_NUM_THREADS} – 1)\)

• \texttt{omp_get_num_threads()} returns the total number of threads involved in the parallel section of the program

• The statement after the parallel directive is executed independently on each of the \( n \) threads.

• If the next C construct is a block (code between \{ and \}) the parallel section begins at the opening curly brace and ends at the close curly brace. After the block is executed, the \( n-1 \) extra threads are deactivated and normal sequential execution begins.
Compiling OpenMP Programs

C:
- Case sensitive directives
- Syntax:
  - #pragma omp directive [clause [clause]…]
- Compiling OpenMP source code:
  - (GNU C compiler): gcc -fopenmp -o exec_file_name file_name.c
  - (Intel C compiler): icc -o exec_file_name -openmp file_name.c

Fortran:
- Case insensitive directives
- Syntax:
  - !$OMP directive [clause[[,] clause]…] (free format)
  - !$OMP / C$OMP / *$OMP directive [clause[[,] clause]…] (free format)
- Compiling OpenMP source code:
  - (GNU Fortran compiler): gfortran -fopenmp -o exec_name file_name.f95
  - (Intel Fortran compiler): ifort -o exe_file_name -openmp file_name.f
Topics

- Introduction to OpenMP
- Runtime library and environment variables
- Data and work sharing directives
- Synchronization
- Reduction
OpenMP: Data Environment

- OpenMP program always begins with a single thread of control – *master thread*
- Context associated with the master thread is also known as the Data Environment.
- Context is comprised of:
  - Global variables
  - Automatic variables
  - Dynamically allocated variables
- Context of the master thread remains valid throughout the execution of the program.
- The OpenMP parallel construct may be used to either share a single copy of the context with all the threads or provide each of the threads with a private copy of the context.
- The sharing of Context can be performed at various levels of granularity:
  - Select variables from a context can be shared while keeping the context private etc.
OpenMP Data Environment

- OpenMP data scoping clauses allow a programmer to decide a variable’s execution context (should a variable be *shared* or *private*.)
- 3 main data scoping clauses in OpenMP (Shared, Private, Reduction):
  - **Shared:**
    - A variable will have a single storage location in memory for the duration of the parallel construct, i.e. references to a variable by different threads access the same memory location.
    - That part of the memory is shared among the threads involved, hence modifications to the variable can be made using simple read/write operations.
    - Modifications to the variable by different threads is managed by underlying shared memory mechanisms.
  - **Private:**
    - A variable will have a separate storage location in memory for each of the threads involved for the duration of the parallel construct.
    - All read/write operations by the thread will affect the thread’s private copy of the variable.
  - **Reduction:**
    - Exhibit both shared and private storage behavior. Usually used on objects that are the target of arithmetic reduction.
    - Example: summation of local variables at the end of a parallel construct.
OpenMP Work-Sharing Directives

- Work sharing constructs divide the execution of the enclosed block of code among the group of threads.
- They do not launch new threads.
- No implied barrier on entry
- Implicit barrier at the end of work-sharing construct
- Commonly used Work Sharing constructs:
  - *for* directive (C/C++; equivalent *DO* construct available in Fortran but will not be covered here) : shares iterations of a loop across a group of threads
  - *sections* directive : breaks work into separate sections between the group of threads; such that each thread independently executes a section of the work.
  - *critical* directive: serializes a section of code
- We will cover *for* and *sections* directive of these in greater detail
OpenMP *for* directive

- *for* directive helps share iterations of a loop between a group of threads
- If *nowait* is specified then the threads do not wait for synchronization at the end of a parallel loop
- The *schedule* clause describes how iterations of a loop are divided among the threads in the team (discussed in detail in the next few slides)

```c
#pragma omp parallel
{
    p=5;
    #pragma omp for
    for (i=0; i<24; i++)
        x[i]=y[i]+p*(i+3)
    ...
    ...
} /* omp end parallel */
```

![Diagram of OpenMP for directive](image)
Simple Loop Parallelization

```c
#pragma omp parallel for
for (i=0; i<n; i++)
    z(i) = a*x(i)+y
```

Master thread executing serial portion of the code

Master thread encounters parallel for loop and creates worker threads

Master and worker threads divide iterations of the for loop and execute them concurrently

Implicit barrier: wait for all threads to finish their executions

Master thread executing serial portion of the code resumes and slave threads are discarded
Example: OpenMP work sharing Constructs

```c
#include <omp.h>
#define N 16
main ()
{
    int i, chunk=1;
    float a[N], b[N], c[N];
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = 4;
    printf("a[i]  +  b[i]  =  c[i] \n");
    #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 */
    for (i=0; i < N; i++)
        printf("%f   +   %f   =   %f \n", a[i], b[i], c[i]);
}
```

- Initializing the vectors `a[i]`, `b[i]`
- Instructing the runtime environment that `a`, `b`, `c`, `chunk` are shared variables and `i` is a private variable
- The `nowait` ensures that the child threads do not synchronize once their work is completed
- Load balancing the threads using a DYNAMIC policy where array is divided into chunks of 4 and assigned to the threads

Modified from examples posted on: https://computing.llnl.gov/tutorials/openMP/
OpenMP *sections* directive

- *sections* directive is a non-iterative work sharing construct.
- Independent *section* of code are nested within a *sections* directive.
- It specifies enclosed *section* of codes between different threads.
- Code enclosed within a *section* directive is executed by a thread within the pool of threads.

```c
#pragma omp parallel private(p)
{
    #pragma omp sections
    {{
        a=...;
        b=...;
    }
#pragma omp section
    {
        p=...;
        q=...;
    }
#pragma omp section
    {
        x=...;
        y=...;
    }
} /* omp end sections */
} /* omp end parallel */
```

Diagram:
- **fork**
- **join**
- Variables: a, b, p, q, x, y
#pragma omp parallel for
for (i=0; i<n; i++)
    z[i] = a*x[i]+y

- Shared variable z is modified by multiple threads
- Each iteration reads the scalar variables a and y and the array element x[i]
- a,y,x can be read concurrently as their values remain unchanged.
- Each iteration writes to a distinct element of z[i] over the index range. Hence write operations can be carried out concurrently with each iteration writing to a distinct array index and memory location
- The parallel for directive in OpenMP ensures that the for loop index value (i in this case) is private to each thread.
Example: OpenMP Sections

```c
#include <omp.h>
#define N 16
main (){  
  int i;  
  float a[N], b[N], c[N], d[N];  
  for (i=0; i < N; i++)  
    a[i] = b[i] = i * 1.5;

#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 */

...  
```

Sections construct that encloses the section calls

Section : that computes the sum of the 2 vectors

Section : that computes the product of the 2 vectors

Modified from examples posted on: https://computing.llnl.gov/tutorials/openMP/
Topics

- Introduction to OpenMP
- Runtime library and environment variables
- Data and work sharing directives
- Synchronization
- Reduction
Thread Synchronization

- “communication” mainly through read write operations on shared variables
- Synchronization defines the mechanisms that help in coordinating execution of multiple threads (that use a shared context) in a parallel program.
- Without synchronization, multiple threads accessing shared memory location may cause conflicts by:
  - Simultaneously attempting to modify the same location
  - One thread attempting to read a memory location while another thread is updating the same location.
- Synchronization helps by providing explicit coordination between multiple threads.
- Two main forms of synchronization:
  - Implicit event synchronization
  - Explicit synchronization – critical, master directives in OpenMP
OpenMP Synchronization: *master*

- The *master* directive in OpenMP marks a block of code that gets executed on a single thread.
- The rest of the threads in the group ignore the portion of code marked by the master directive.
- Example

  ```
  #pragma omp master
  structured block
  ```

*Race Condition:*

Two asynchronous threads access the same shared variable and at least one modifies the variable and the sequence of operations is undefined. Result of these asynchronous operations depends on detailed timing of the individual threads of the group.
OpenMP *critical* directive: Explicit Synchronization

- Race conditions can be avoided by controlling access to shared variables by allowing threads to have exclusive access to the variables.
- Exclusive access to shared variables allows the thread to *atomically* perform read, modify and update operations on the variable.
- *Mutual exclusion* synchronization is provided by the *critical* directive of OpenMP.
- Code block within the *critical region* defined by *critical* / *end critical* directives can be executed only by one thread at a time.
- Other threads in the group must wait until the current thread exits the critical region. Thus only one thread can manipulate values in the critical region.

```c
int x
x=0;
#pragma omp parallel shared(x)
{
    #pragma omp critical
    x = 2*x + 1;
}
/* omp end parallel */
```
Simple Example: \textit{critical}

cnt = 0;
f = 7;
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<20; i++)
    {
        if (b[i] == 0)
        {
            #pragma omp critical
            cnt++;
        }
        a[i] = b[i] + f*(i+1);
    }
} /* omp end parallel */
Topics

• Introduction to OpenMP
• Runtime library and environment variables
• Data and work sharing directives
• Synchronization
• Reduction
OpenMP: Reduction

- performs reduction on *shared variables* in list based on the *operator* provided.
- for C/C++ operator can be any one of:
  - +, *, -, ^, |, ||, & or &&
  - At the end of a reduction, the shared variable contains the result obtained upon combination of the list of variables processed using the operator specified.

```
sum = 0.0
#pragma omp parallel for reduction(+:sum)
for (i=0; i < 20; i++)
    sum = sum + (a[i] * b[i]);
```
Example: Reduction

```c
#include <omp.h>
main () {
  int i, n, chunk;
  float a[16], b[16], result;
  n = 16;
  chunk = 4;
  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);
}
```

Reduction example with summation where the result of the reduction operation stores the dotproduct of two vectors $\sum a[i]*b[i]$

SRC: https://computing.llnl.gov/tutorials/openMP/