Shared-Memory Programming in OpenMP

Advanced Research Computing

www.arc.vt.edu
Outline

• What is OpenMP?
• How does OpenMP work?
  – Architecture
  – Fork-join model of parallelism
  – Communication
• OpenMP constructs
  – Directives
  – Runtime Library API
  – Environment variables
Overview
What is OpenMP?

• API for parallel programming on shared memory systems
  – Parallel “threads”

• Implemented through the use of:
  – Compiler Directives
  – Runtime Library
  – Environment Variables

• Supported in C, C++, and Fortran

• Maintained by OpenMP Architecture Review Board (http://www.openmp.org/)
Advantages

• Code looks similar to sequential
  – Relatively easy to learn
  – Adding parallelization can be incremental

• No message passing

• Coarse-grained or fine-grained parallelism

• Widely-supported
Disadvantages

- Scalability limited by memory architecture
  - To a single node (8 to 32 cores) on most machines
- Managing shared memory can be tricky
- Improving performance is not always guaranteed or easy
• Your laptop
• Multicore, multiple memory NUMA system
  – HokieOne (SGI UV)
• One node on a hybrid system
• Parallelism by region
• Master Thread: Initiated at run-time & persists throughout execution
  – Assembles team of parallel threads at parallel regions
How do threads communicate?

• Every thread has access to “global” memory (shared). Each thread has access to a stack memory (private).

• Use shared memory to communicate between threads.

• Simultaneous updates to shared memory can create a race condition. Results change with different thread scheduling.

• Use mutual exclusion to avoid data sharing - but don’t use too many because this will serialize performance.
Race Conditions

Example: Two threads ("T1" & "T2") increment $x=0$

Start: $x=0$
1. T1 reads $x=0$
2. T1 calculates $x=0+1=1$
3. T1 writes $x=1$
4. T2 reads $x=1$
5. T2 calculates $x=1+1=2$
6. T2 writes $x=2$

Result: $x=2$

Start: $x=0$
1. T1 reads $x=0$
2. T2 reads $x=0$
3. T1 calculates $x=0+1=1$
4. T2 calculates $x=0+1=1$
5. T1 writes $x=1$
6. T2 writes $x=1$

Result: $x=1$
OpenMP Basics
OpenMP language extensions

- **parallel control structures**
  - Governs flow of control in the program
  - *parallel* directive

- **work sharing**
  - Distributes work among threads
  - *do/parallel do* and *Section* directives

- **data environment**
  - Specifies variables as shared or private
  - *shared* and *private* clauses

- **synchronization**
  - Coordinates thread execution
  - *critical* and *atomic* directives
  - *barrier* directive

- **runtime functions, env. variables**
  - Runtime environment
  - *omp_set_num_threads()*
  - *omp_get_thread_num()*
  - *OMP_NUM_THREADS*
  - *OMP_SCHEDULE*
OpenMP directives specify parallelism within source code:

- **C/C++:** directives begin with the `# pragma omp` sentinel.
- **FORTRAN:** Directives begin with the `!$OMP, C$OMP` or `*$OMP` sentinel.
- **F90:** `!$OMP` free-format

- Parallel regions are marked by enclosing parallel directives
- Work-sharing loops are marked by parallel do/for

```fortran
!$OMP parallel
  ...
!$OMP end parallel

!$OMP parallel do
  DO ...
!$OMP end parallel do
```

```c/c++
#pragma omp parallel
{...}
#pragma omp parallel for
for(){...}
```
### API: Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_get_num_threads()</code></td>
<td>Returns number of threads in team</td>
</tr>
<tr>
<td><code>omp_get_thread_num()</code></td>
<td>Returns thread ID (0 to n-1)</td>
</tr>
<tr>
<td><code>omp_get_num_procs()</code></td>
<td>Returns number of machine CPUs</td>
</tr>
<tr>
<td><code>omp_in_parallel()</code></td>
<td>True if in parallel region &amp; multiple threads executing</td>
</tr>
<tr>
<td><code>omp_set_num_threads(#)</code></td>
<td>Changes number of threads for parallel region</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_get_dynamic()</code></td>
<td>True if dynamic threading is on</td>
</tr>
<tr>
<td><code>omp_set_dynamic()</code></td>
<td>Set state of dynamic threading (true/false)</td>
</tr>
</tbody>
</table>
API: Environment Variables

• **OMP_NUM_THREADS**: Number of Threads

• **OMP_DYNAMIC**: TRUE/FALSE for enable/disable dynamic threading
Parallel Regions

1  !$OMP PARALLEL
2    code block
3    call work(...)
4  !$OMP END PARALLEL

•Line 1: Team of threads formed at parallel region
•Lines 2-3:
  – Each thread executes code block and subroutine calls
  – No branching (in or out) in a parallel region
•Line 4: All threads synchronize at end of parallel region (implied barrier).
Example: Hello World

• Update a serial code to run on multiple cores using OpenMP

1. Start from serial “Hello World” example:
   • hello.c, hello.f

2. Create a parallel region

3. Identify individual threads and print out information from each
Hello World in OpenMP

Fortran:

```fortran
!$OMP PARALLEL
  INTEGER tid
  tid = OMP_GET_THREAD_NUM()
  PRINT *, 'Hello from thread = ', tid
!$OMP END PARALLEL
```

C:

```c
#pragma omp parallel
{
  int tid;
  tid = omp_get_thread_num();
  printf(`Hello from thread =%d\n`, tid);
}
```
Compiling with OpenMP

• GNU uses –fopenmp flag
  gcc program.c -fopenmp -o runme
  g++ program.cpp -fopenmp -o runme
  gfortran program.f -fopenmp -o runme

• Intel uses –openmp flag, e.g.
  icc program.c -openmp -o runme
  ifort program.f -openmp -o runme
OpenMP Constructs
Use OpenMP directives to specify Parallel Region and Work-Sharing constructs.

- **Parallel**
  - **End Parallel**
  - Parallel DO/for
  - Parallel SECTIONS

<table>
<thead>
<tr>
<th>Code block</th>
<th>Each Thread Executes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO</td>
<td>Work Sharing</td>
</tr>
<tr>
<td>SECTIONS</td>
<td>Work Sharing</td>
</tr>
<tr>
<td>SINGLE</td>
<td>One Thread</td>
</tr>
<tr>
<td>MASTER</td>
<td>Only the master thread</td>
</tr>
<tr>
<td>CRITICAL</td>
<td>One Thread at a time</td>
</tr>
</tbody>
</table>

Stand-alone

Parallel Constructs
OpenMP parallel constructs

- **Replicated**

  ```
  PARALLEL
  {code}
  END PARALLEL
  ```

  ![Replicated Diagram]

- **Work Sharing**

  ```
  PARALLEL DO
  do I = 1,N*4
  {code}
  end do
  END PARALLEL DO
  ```

  ![Work Sharing Diagram]

- **Combined**

  ```
  PARALLEL
  {code1}
  DO
  do I = 1,N*4
  {code2}
  end do
  END DO
  {code3}
  END PARALLEL
  ```

  ![Combined Diagram]
There are two OpenMP “modes”

- **static** mode
  - Fixed number of threads
- **dynamic** mode:
  - Number of threads can change under user control from one parallel region to another (using `OMP_set_num_threads`)
  - Specified by setting an environment variable
    
    (csh)  setenv OMP_DYNAMIC true
    (bash) export OMP_DYNAMIC=true

*Note*: the user can only define the maximum number of threads, compiler can use a smaller number
Parallel Constructs

- **PARALLEL**: Create threads, any code is executed by all threads
- **DO/FOR**: Work sharing of iterations
- **SECTIONS**: Work sharing by splitting
- **SINGLE**: Only one thread
- **CRITICAL** or **ATOMIC**: One thread at a time
- **MASTER**: Only the master thread
The DO / for directive

Fortran:

```fortran
!$OMP PARALLEL DO
  do i=0,N
C    do some work
   enddo
!$OMP END PARALLEL DO
```

C:

```c
#pragma omp parallel for
{
  for (i=0; i<N; i++)
    // do some work
}
```
The DO / for Directive

1  !$OMP PARALLEL DO
2     do i=1,N
3       a(i) = b(i) + c(i)
4     enddo
5  !$OMP END PARALLEL DO

Line 1  Team of threads formed (parallel region).
Line 2-4 Loop iterations are split among threads.
Line 5 (Optional) end of parallel loop (implied barrier at enddo).

Each loop iteration must be independent of other iterations.
The Sections Directive

• Different threads will execute different code
• Any thread may execute a section

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            // do some work
        } // end of section
        #pragma omp section
        {
            // do some different work
        } // end of section
    } // end of sections
} // end of parallel region
```
The !$OMP PARALLEL directive declares an entire region as parallel. Merging work-sharing constructs into a single parallel region eliminates the overhead of separate team formations.

```
!$OMP PARALLEL
  !$OMP DO
    do i=1,n
      a(i)=b(i)+c(i)
    enddo
  !$OMP END DO
!$OMP DO
  do i=1,m
    x(i)=y(i)+z(i)
  enddo
!$OMP END DO
!$OMP END PARALLEL
```

```
!$OMP PARALLEL DO
do i=1,n
  a(i)=b(i)+c(i)
enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
do i=1,m
  x(i)=y(i)+z(i)
enddo
!$OMP END PARALLEL DO
!$OMP END PARALLEL DO
```
OpenMP clauses

Control the behavior of an OpenMP directive:

• Data scoping (Private, Shared, Default)
• Schedule (Guided, Static, Dynamic, etc.)
• Initialization (e.g. COPYIN, FIRSTPRIVATE)
• Whether to parallelize a region or not (if-clause)
• Number of threads used (NUM_THREADS)
Private and Shared Data

- **Shared**: Variable is shared (seen) by all processors.
- **Private**: Each thread has a private instance (copy) of the variable.
- **Defaults**: All DO LOOP indices are private, all other variables are shared.

```c
!$OMP PARALLEL DO SHARED(A,B,C,N)
PRIVATE(i)
  do i=1,N
    A(i) = B(i) + C(i)
  enddo
!$OMP END PARALLEL DO
```
• In the following loop, each thread needs its own PRIVATE copy of TEMP.

• If TEMP were shared, the result would be unpredictable since each processor would be writing and reading to/from the same memory location.

```fortran
!$OMP PARALLEL DO SHARED(A,B,C,N) PRIVATE(temp,i)
do i=1,N
    temp = A(i)/B(i)
    C(i) = temp + cos(temp)
endo
!$OMP END PARALLEL DO
```

• A lastprivate(temp) clause will copy the last loop(stack) value of temp to the (global) temp storage when the parallel DO is complete.

• A firstprivate(temp) would copy the global temp value to each stack’s temp.
int tid, pr=-1, fp=-1, sh=-1, df=-1;
printf("BEGIN: pr is %d, fp is %d, sh is %d, df is %d.
\n",pr,fp,sh,df);

#pragma omp parallel shared(sh) private(pr,tid) firstprivate(fp)
{
  tid = omp_get_thread_num();
  printf("Thread %d START : pr is %d, fp is %d, sh is %d, df is %d.
\n",tid,pr,fp,sh,df);

  pr = tid * 4; fp = pr; sh = pr; df = pr;
  printf("Thread %d UPDATE: pr is %d, fp is %d, sh is %d, df is %d.
\n",tid,pr,fp,sh,df);
}
/* end of parallel section */

printf("END: pr is %d, fp is %d, sh is %d, df is %d.
\n",pr,fp,sh,df);
Data Scoping Example (Code)

```bash
$ icc -openmp omp_scope.c -o omp_scope
$ ./omp_scope
BEGIN: pr is -1, fp is -1, sh is -1, df is -1.
Thread 0 START : pr is 0, fp is -1, sh is -1, df is -1.
Thread 1 START : pr is 0, fp is -1, sh is -1, df is -1.
Thread 1 UPDATE: pr is 4, fp is 4, sh is 4, df is 4.
Thread 2 START : pr is 0, fp is -1, sh is -1, df is -1.
Thread 2 UPDATE: pr is 8, fp is 8, sh is 8, df is 8.
Thread 0 UPDATE: pr is 0, fp is 0, sh is 0, df is 0.
Thread 3 START : pr is 0, fp is -1, sh is 8, df is 8.
Thread 3 UPDATE: pr is 12, fp is 12, sh is 12, df is 12.
END: pr is -1, fp is -1, sh is 12, df is 12.
```
!OMP$ PARALLEL DO SCHEDULE(STATIC)
Each CPU receives one set of contiguous iterations (~total_no_iterations / no_of_cpus).

!OMP$ PARALLEL DO SCHEDULE(STATIC,C)
Iterations are divided round-robin fashion in chunks of size C.

!OMP$ PARALLEL DO SCHEDULE(DYNAMIC,C)
Iterations handed out in chunks of size C as CPUs become available.

!OMP$ PARALLEL DO SCHEDULE(GUIDED,C)
Each of the iterations are handed out in pieces of exponentially decreasing size, with C minimum number of iterations to dispatch each time (Important for load balancing.)
Load Imbalances

Thread 0
Thread 1
Thread 2
Thread 3

Unused Resources

Time

www.arc vt.edu
Example - SCHEDULE(STATIC,16)

```c
!$OMP parallel do schedule(static,16)
    do i=1,128
        A(i)=B(i)+C(i)
    enddo
!$OMP_NUM_THREADS=4
do i=1,128
    A(i)=B(i)+C(i)
enddo

code for thread0:
do i=1,16
    A(i)=B(i)+C(i)
enddo
do i=65,80
    A(i)=B(i)+C(i)
enddo

code for thread1:
do i=17,32
    A(i)=B(i)+C(i)
enddo
do i=81,96
    A(i)=B(i)+C(i)
enddo

code for thread2:
do i=33,48
    A(i)=B(i)+C(i)
enddo
do i=97,112
    A(i)=B(i)+C(i)
enddo

code for thread3:
do i=49,64
    A(i)=B(i)+C(i)
enddo
do i=113,128
    A(i)=B(i)+C(i)
enddo
```
### Static

**PROS**
- Low compute overhead
- No synchronization overhead per chunk
- Takes better advantage of data locality

**CONS**
- Cannot compensate for load imbalance

### Dynamic

**PROS**
- Potential for better load balancing, especially if chunk is low

**CONS**
- Higher compute overhead
- Synchronization cost associated per chunk of work
Scheduling Options

• When shared array data is reused multiple times, prefer static scheduling to dynamic

• Every invocation of the scaling would divide the iterations among CPUs the same way for static but not so for dynamic scheduling

```c
!$OMP parallel private (i,j,iter)
do iter=1,niter
...
!$OMP do
do j=1,n
  do i=1,n
    A(i,j)=A(i,j)*scale
  end do
end do
...
doi
!$OMP end parallel
```
### Comparison of scheduling options

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Chunk</th>
<th>Chunk Size</th>
<th>Chunk #</th>
<th>Static or Dynamic</th>
<th>Compute Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Static</td>
<td>Simple</td>
<td>No</td>
<td>N/P</td>
<td>P</td>
<td>Static</td>
<td>Lowest</td>
</tr>
<tr>
<td>Interleaved</td>
<td>Simple</td>
<td>Yes</td>
<td>C</td>
<td>N/C</td>
<td>Static</td>
<td>Low</td>
</tr>
<tr>
<td>Simple Dynamic</td>
<td>Dynamic</td>
<td>Optional</td>
<td>C</td>
<td>N/C</td>
<td>Dynamic</td>
<td>Medium</td>
</tr>
<tr>
<td>Guided</td>
<td>Guided</td>
<td>Optional</td>
<td>Decreasing from N/P</td>
<td>Fewer than N/C</td>
<td>Dynamic</td>
<td>High</td>
</tr>
<tr>
<td>Runtime</td>
<td>Runtime</td>
<td>No</td>
<td>Varies</td>
<td>Varies</td>
<td>Varies</td>
<td>Varies</td>
</tr>
</tbody>
</table>
/*** Initialize matrices ***/
for (i=0; i<NRA; i++)
  for (j=0; j<NCA; j++)
    a[i][j] = i+j;
[etc...also initialize b and c]

/*** Multiply matrices ***/
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];
Example: Matrix Multiplication

Parallelize matrix multiplication from serial:
  - C version: mm.c
  - Fortran version: mm.f

1. Use OpenMP to parallelize loops
2. Determine public / private variables
3. Decide how to schedule loops
/***/ Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
{
  tid =omp_get_thread_num();

  /***/ 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<NRA; i++)
    { 
    printf("Thread=%d did row=%d\n",tid,i);
    for(j=0; j<NCB; j++)
      for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
}
Matrix Multiplication: Work Sharing

• Partition by rows:

\[
\text{\input{diagram}}
\]
Reduction Clause

- Thread-safe way to combine private copies of a variable into a single result
- Variable that accumulates the result is the “reduction variable”
- After loop execution, master thread collects private values of each thread and finishes the (global) reduction
- Reduction operators and variables must be declared
```c
double n_sqr=0;  //square of the vector norm

#pragma omp parallel shared(vec, dim) private(i)  //create threads
{
    //Split up the for loop
    //Use the reduction() clause to have OpenMP
    //sum up all of the private copies of n_sqr
    #pragma omp for reduction(+:n_sqr)
    for(i=0; i<dim; i++)  //iterate through the rows of the result
        n_sqr += vec[i]*vec[i];
}

printf("Done.
Vector norm is %.5f.
\n",sqrt(n_sqr));
```
SYNCHRONIZATION
• When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.
• By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```c
$OMP PARALLEL
$OMP DO
    do i=1,n
        work(i)
    enddo
$OMP END DO NOWAIT
$OMP DO schedule(dynamic,M)
    do i=1,m
        x(i)=y(i)+z(i)
    enddo
$OMP END DO
$OMP END PARALLEL
```
• Create a barrier to synchronize threads

```c
#pragma omp parallel
{
    // all threads do some work
    #pragma omp barrier
    // all threads do more work
}
```

• Barrier is implied at the end of a parallel region
Mutual Exclusion: Critical/Atomic Directives

- **ATOMIC** For a single command (e.g. incrementing a variable)
- **CRITICAL** Directive: Longer sections of code

```c
!$OMP PARALLEL SHARED(sum,X,Y)
...
!$OMP ATOMIC
    sum=sum+1
...
!$OMP END PARALLEL

!$OMP PARALLEL SHARED(sum,X,Y)
...
!$OMP CRITICAL
    call update(x)
    call update(y)
    sum=sum+1
!$OMP END CRITICAL
...
!$OMP END PARALLEL
```
Mutual exclusion: lock routines

When each thread must execute a section of code serially, locks provide a more flexible way of ensuring serial access than CRITICAL and ATOMIC directives.

```plaintext
call OMP_INIT_LOCK(maxlock)
!$OMP PARALLEL SHARED(X,Y)
...
call OMP_set_lock(maxlock)
call update(x)
call OMP_unset_lock(maxlock)
...
!$OMP END PARALLEL
call OMP_DESTROY_LOCK(maxlock)
```
Performance Optimization
**OpenMP wallclock timers**

Real*8 :: omp_get_wtime, omp_get_wtick() (Fortran)

double omp_get_wtime(), omp_get_wtick(); (C)

double t0, t1, dt, res;
...

t0 = omp_get_wtime();
<work>
t1 = omp_get_wtime();
dt = t1 - t0;
res = 1.0/omp_get_wtick();
printf(“Elapsed time = %lf\n”,dt);
printf(“clock resolution = %lf\n”,res);
#Case 1:
Normal C Timer: 0.230 seconds
OpenMP Timer: 0.105319 seconds

#Case 2 (more efficient threading):
Normal C Timer: 0.200 seconds
OpenMP Timer: 0.012919 seconds
• All processors connected to same memory and all memory is identical
• Ithaca node (8 cores, 24 GB memory):
  - 2 sockets with 4 cores each
  - 32 KB L1 cache, 256KB L2 cache, 8MB L3 cache
Cache Topology

Level: 1
Size: 32 kB
Cache groups: 0 1 2 3 4 5 6 7

Level: 2
Size: 256 kB
Cache groups: 0 1 2 3 4 5 6 7

Level: 3
Size: 8 MB
Cache groups: 0 1 2 3 4 5 6 7
NUMA Topology

NUMA domains: 2

Domain 0:
Processors: 0 1 2 3
Relative distance to nodes: 10 21
Memory: 10444.7 MB free of total 12277.5 MB

Domain 1:
Processors: 4 5 6 7
Relative distance to nodes: 21 10
Memory: 11551.6 MB free of total 12288 MB
OpenMP and cc-NUMA

- cc-NUMA = cache coherent non-uniform memory access
- Modern CPU’s utilize multiple levels of cache to reduce the time to get data to the ALU
OpenMP and cc-NUMA

- Setup is advantageous because it allows individual CPU cores to get data more quickly from memory.
- Maintaining cache coherence is expensive.
- Result: you want to associate specific memory to specific CPU cores.
OpenMP and cc-NUMA

1. Bind specific threads to specific cores:
   Intel: export KMP_AFFINITY="proclist=[$CPUSET]"
   GCC: export GOMP_CPU_AFFINITY="$CPUSET"

2. Associate memory with a specific thread:
   - First-touch policy: use parallel initialization so that values are initialized by the thread that will modify the value
//serial initialization:
//OS will allocate all data close to initial thread
for (i=0;i<N;i++) a[i]=b[i]=c[i]=0.0;

//saxpying with poor memory placement
#pragma omp parallel for
for(i=0;i<N;i++) a[i]=b[i]+scalar*c[i];

//parallel initialization: data allocated where used
#pragma omp parallel for
for (i=0;i<N;i++) a[i]=b[i]=c[i]=0.0;

//saxpying with optimal memory placement
#pragma omp parallel for
for(i=0;i<N;i++) a[i]=b[i]+scalar*c[i];
$ icc -openmp omp_saxpy.c -o saxpy
$ ./saxpy
SAXPY with Serial Initialization:
Elapsed time = 0.105552

SAXPY with Parallel Initialization:
Elapsed time = 0.012795
Online Content

• ARC OpenMP page: http://www.arc.vt.edu/openmp
• OpenMP Application Programming Interface: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
• LLNL Examples: https://computing.llnl.gov/tutorials/openMP/exercise.html
Thank You!

www.arc.vt.edu