Shared-Memory Programming in OpenMP
Advanced Research Computing

www.arc.vt.edu
• What is OpenMP?
• How does OpenMP work?
  – Architecture
  – Fork-join model of parallelism
  – Communication
• OpenMP constructs
  – Directives
  – Runtime Library API
  – Environment variables
• Login using temporary accounts on Ithaca
  • Username and password at your desk
  ssh hpcXX@ithaca2.arc.vt.edu

• Copy example code to your home directory
  cp –r /archive/groups/training/
  OpenMP_EXAMPLES .

• More examples can be downloaded:
  • https://computing.llnl.gov/tutorials/openMP/exercise.html
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
• 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
  - parallel directive
  - governs flow of control in the program
- Work sharing
  - do/parallel do and Section directives
  - distributes work among threads
- Data environment
  - shared and private clauses
- Synchronization
  - critical and atomic directives
  - coordinates thread execution
  - barrier directive
- Runtime functions, env. variables
  - 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>omp_get_num_threads()</td>
<td>Returns number of threads in team</td>
</tr>
<tr>
<td>omp_get_thread_num()</td>
<td>Returns thread ID (0 to n-1)</td>
</tr>
<tr>
<td>omp_get_num_procs()</td>
<td>Returns number of machine CPUs</td>
</tr>
<tr>
<td>omp_in_parallel()</td>
<td>True if in parallel region &amp; multiple threads executing</td>
</tr>
<tr>
<td>omp_set_num_threads(#)</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>omp_get_dynamic()</td>
<td>True if dynamic threading is on.</td>
</tr>
<tr>
<td>omp_set_dynamic()</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
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
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);
}
```
OpenMP Constructs
Parallel Region/Work Sharing

Use OpenMP directives to specify Parallel Region and Work-Sharing constructs.

Parallel

End Parallel

Parallel DO/for
Parallel SECTIONS

Code block | Each Thread Executes
-----------|---------------------
DO         | Work Sharing
SECTIONS   | Work Sharing
SINGLE     | One Thread
MASTER     | Only the master thread
CRITICAL   | One Thread at a time

Stand-alone Parallel Constructs

www.arc.vt.edu
OpenMP parallel constructs

PARALLEL
{code}
END PARALLEL

Replicated

do I = 1, N
{code}
end do

Work Sharing

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

Combined

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

www.arc.vt.edu
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
}
```
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 }
   #pragma omp section
   { // do some different work }
 } // 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.
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
```
Private data example

• 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.

```c
!$OMP PARALLEL DO SHARED(A,B,C,N) PRIVATE(temp,i)
   do i=1,N
      temp = A(i)/B(i)
      C(i) = temp + cos(temp)
   enddo
!$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.
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)
!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
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

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

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

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

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
...
end do
!$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>

[Source: www.arc.vt.edu](http://www.arc.vt.edu)
Loop Collapse

• Allow collapsing of perfectly nested loops

• Will form a single loop and then parallelize it:

\[
\begin{align*}
!$omp \parallel do \ do \ collapse(2) \\
do \ i=1,n \\
 \quad do \ j=1,n \\
 \quad \quad \quad \quad \ldots \\
 \quad \quad end \ do \\
end \ do
\end{align*}
\]
Example: Matrix Multiplication

Serial Algorithm:

```c
void mat_mult(double *mat1, double *mat2, double *result, int dim)
{
    int i,j,k; //iterators

    for(i=0; i<dim; i++) //iterate through the rows of the result
        for(j=0; j<dim; j++) //iterate through the columns of the result
        {
            *( result+(j+i*dim) ) = 0; //initialize

            //iterate through the inner dimension (columns of first matrix/rows of second)
            for(k=0; k<dim; k++)
                *( result+(j+i*dim) ) += *( mat1+(k+i*dim) )*( *( mat2+(j+k*dim) ) );
        }
} //end mat_mult()
```
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
Matrix Multiplication - OpenMP

```c
void mat_mult_thr(double *mat1, double *mat2, double *result, int dim, int nthr)
{
    int part_rows = dim/nthr; // rows per thread
    omp_set_num_threads(nthr); // set the number of threads

    #pragma omp parallel shared(mat1,mat2,result,part_rows) // create threads
    {
        int i,j,k; // iterators

        #pragma omp for schedule(guided,part_rows) // split up the for loop
        for(i=0; i<dim; i++) // iterate through the rows of the result
        {
            for(j=0; j<dim; j++) // iterate through the columns of the result
            {
                *( result+(j+i*dim) ) = 0; // initialize

                // iterate through the inner dimension (columns of first matrix/rows of second mat)
                for(k=0; k<dim; k++)
                    *( result+(j+i*dim) ) += *( mat1+(k+i*dim) )*(*( mat2+(j+k*dim) ));
            }
        }
    } // end mat_mult_thr()
```
• Partition by rows:
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
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.\nVector norm is %.5f.\n\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
• 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
```

```c
!$omp parallel shared(sum,X,Y)
...
!$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.

```fortran
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
", dt);
  printf("clock resolution = %lf\n", res);
Shared memory to OpenMP

• All processors connected to same memory and all memory is identical

- Diagram showing five processors (P) connected to a single memory block.
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
• 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=[CPUS]
   GCC: export GOMP_CPU_AFFINITY="CPUS"

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
Simple Example, Part 2

DAXPY with OpenMP (65,536 iterations)

- Static Schedule
- Guided Schedule

Time (seconds)

Vector Length

www.arc.vt.edu
Exercise: Performance Optimization

1. Insert a timer into your OpenMP code
2. Run the code using 1, 2, 4, 8 cores – does your code scale as you increase the number of cores?
3. Can you make the code run faster?
   – Improve load balancing
   – Reduce overhead
   – Parallel initialization
• 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!