Introduction to OpenMP

Alexander B. Pacheco

User Services Consultant
LSU HPC & LONI
sys-help@loni.org

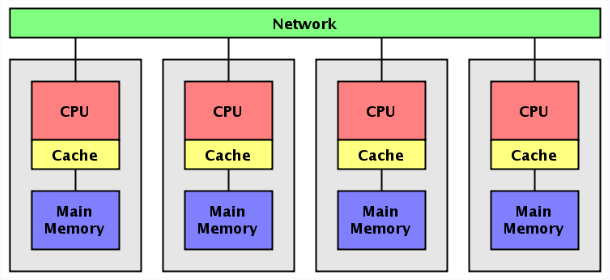
LONI Workshop: Fortran Programming
Louisiana State University
Baton Rouge
Feb 13-16, 2012

Goals

- Acquaint users with the concepts of shared memory parallelism.
- Acquaint users with the basics of programming with OpenMP.

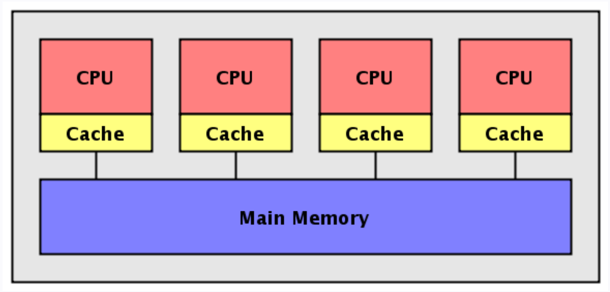
Distributed Memory Model

- Each process has its own address space
  - Data is local to each process
- Data sharing is achieved via explicit message passing
- Example
  - MPI



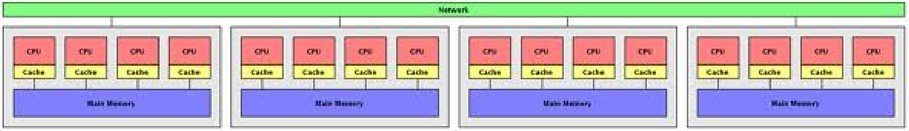
Shared Memory Model

- All threads can access the global memory space.
- Data sharing achieved via writing to/reading from the same memory location
- Example
  - OpenMP
  - Pthreads



Clusters of SMP nodes

- The shared memory model is most commonly represented by Symmetric Multi-Processing (SMP) systems
  - Identical processors
  - Equal access time to memory
- Large shared memory systems are rare, clusters of SMP nodes are popular.



Shared vs Distributed

Shared Memory

- Pros
  - Global address space is user friendly
  - Data sharing is fast
- Cons
  - Lack of scalability
  - Data conflict issues

Distributed Memory

- Pros
  - Memory scalable with number of processors
  - Easier and cheaper to build
- Cons
  - Difficult load balancing
  - Data sharing is slow

OpenMP

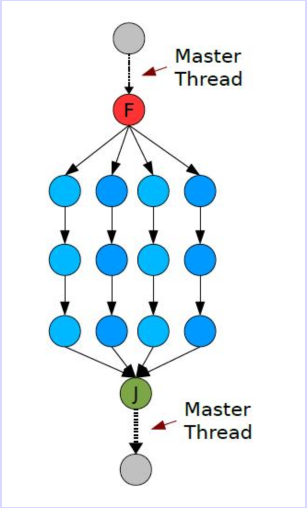
- OpenMP is an Application Program Interface (API) for thread based parallelism; Supports Fortran, C and C++
- Uses a fork-join execution model
- OpenMP structures are built with program directives, runtime libraries and environment variables
- OpenMP has been the industry standard for shared memory programming over the last decade
  - Permanent members of the OpenMP Architecture Review Board: AMD, Cray, Fujutsu, HP, IBM, Intel, Microsoft, NEC, PGI, SGI, Sun
- OpenMP 3.1 was released in September 2011

Advantages of OpenMP

- Portability
  - Standard among many shared memory platforms
  - Implemented in major compiler suites
- Ease to use
  - Serial programs can be parallelized by adding compiler directives
  - Allows for incremental parallelization - a serial program evolves into a parallel program by parallelizing different sections incrementally

Fork-Join Execution Model

- Parallelism is achieved by generating multiple threads that run in parallel
  - A fork is when a single thread is made into multiple, concurrently executing threads
  - A join is when the concurrently executing threads synchronize back into a single thread
- OpenMP programs essentially consist of a series of forks and joins.



Building Block of OpenMP

- Program directives
    - Syntax
        - C/C++: `#pragma omp <directive> [clause]`
        - Fortran: `!$omp <directive> [clause]`
    - Parallel regions
    - Parallel loops
    - Synchronization
    - Data Structure
    - ...
- Runtime library routines
- Environment variables

Hello World: C

```
#include <omp.h>
#include <stdio.h>
int main () {
  #pragma omp parallel
  {
    printf("Hello from thread %d out of %d
      threads\n",omp_get_thread_num(),
      omp_get_num_threads());
  }
  return 0;
}
```

OpenMP include file

Parallel region starts here

Runtime library functions

Parallel region ends here

Output

```
Hello from thread 0 out of 4 threads
Hello from thread 1 out of 4 threads
Hello from thread 2 out of 4 threads
Hello from thread 3 out of 4 threads
```

```
program hello

   implicit none
   integer :: omp_get_thread_num, omp_get_num_threads

   !$omp parallel

   print *, 'Hello from thread',omp_get_thread_num() , &
      'out of ' omp_get_num_threads(),' threads'

   !$omp end parallel
end program hello
```

Parallel region starts here

Runtime library functions

Parallel region ends here

Output

```
Hello from thread 0 out of 4 threads
Hello from thread 1 out of 4 threads
Hello from thread 2 out of 4 threads
Hello from thread 3 out of 4 threads
```

Compilation and Execution

- IBM Power5 and Power7 clusters
  - Use thread-safe compilers (with "_r")
  - Use '-qsmp=omp' option

```
% xlc_r -qsmp=omp hello.c && OMP_NUM_THREADS=4 ./a.out
% xlf90_r -qsmp=omp hello.f90 && OMP_NUM_THREADS=4 ./a.out
```

- Dell Linux clusters
  - Use '-openmp' option (Intel compiler)
- % icc -openmp hello.c && OMP_NUM_THREADS=4 ./a.out
- % ifort -openmp hello.f90 && OMP_NUM_THREADS=4 ./a.out

Exercise 1: Hello World

- Write a “hello world” program with OpenMP where
  1. If the thread id is odd, then print a message "Hello world from thread x, I’m odd!”
  2. If the thread id is even, then print a message "Hello world from thread x, I’m even!”

Solution

C/C++

```
#include <omp.h>
#include <stdio.h>
int main() {
  int id;
#pragma omp parallel private(id)
  {
    id = omp_get_thread_num();
    if (id%2==1)
      printf("Hello world from
thread %d, I am odd\n", id);
    else
      printf("Hello world from
thread %d, I am even\n", id);
  }
}
```

Fortran

```
program hello
  implicit none
  integer i,omp_get_thread_num
  !$omp parallel private(i)
  i = omp_get_thread_num()
  if (mod(i,2).eq.1) then
     print *,'Hello world from
     thread',i,', I am odd!'
  else
     print *,'Hello world from
     thread',i,', I am even!'
  endif
  !$omp end parallel
end program hello
```

Work Sharing: Parallel Loops

- We need to share work among threads to achieve parallelism
- Loops are the most likely targets when parallelizing a serial program
- Syntax:
  - Fortran: `!$omp parallel do`
  - C/C++: `#pragma omp parallel for`
- Other work sharing directives available
  - Sections
  - Tasks

Example: Parallel Loops

C/C++

```
#include <omp.h>
int main() {
    int i=0,N=100,a[100] ;
    #pragma omp parallel for
    for (i=0;i<N;i++) {
        a[i]=some_function(i) ;
    }
}
```

Fortran

```
program paralleldo
    implicit none
    integer i,n,a(100)
    i= 0
    n = 100
    !$omp parallel do
    do i=1,n
        a(i) = some_function(i)
    end do
    !$omp end parallel do
end program paralleldo
```

Load Balancing I

- OpenMP provides different methods to divide iterations among threads, indicated by the `schedule` clause
  - Syntax: `schedule (<method>, [chunk size])`
- Methods include
  - `Static`: the default schedule; divide interations into chunks according to `size`, then distribute chunks to each thread in a round-robin manner.
  - `Dynamic`: each thread grabs a chunk of iterations, then requests another chunk upon completion of the current one, until all iterations are executed.
  - `Guided`: similar to `Dynamic`; the only difference is that the chunk size starts large and shrinks to `size` eventually.

*Load Balancing **II***

4 threads, 100 iterations

Schedule	Iterations mapped onto thread 0	1	2	3
`Static`	1-25	26-50	51-75	76-100
`Static,20`	1-20, 81-100	21-40	41-60	61-80
`Dynamic`	$1,\cdots$	$2,\cdots$	$3,\cdots$	$4,\cdots$
`Dynamic,10`	$1-10,\cdots$	$11-20,\cdots$	$21-30,\cdots$	$31-40,\cdots$

Load Balancing III

Schedule	When to Use
`Static`	Even and predictable workload per iteration; scheduling may be done at compilation time, least work at runtime.
`Dynamic`	Highly variable and unpredictable workload per iteration; most work at runtime
`Guided`	Special case of `dynamic` scheduling; compromise between load balancing and scheduling overhead at runtime

Work Sharing: Sections

- Gives a different block to each thread

C/C++

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
         some_calculation() ;
    #pragma omp section
         some_more_calculation() ;
    #pragma omp section
         yet_some_more_calculation() ;
  }
}
```

Fortran

```
!$omp parallel
  !$omp sections
  !$omp section
     call some_calculation
  !$omp section
     call some_more_calculation
  !$omp section
     call yet_some_more_calculation
  !$omp end sections
!$omp end parallel
```

Scope of variables

- `Shared(list)`
  - Specifies the variables that are shared among all threads
- `Private(list)`
  - Creates a local copy of the specified variables for each thread
  - the value is uninitialized!
- `Default(shared|private|none)`
  - Defines the default scope of variables
  - **C/C++ API does not have `default(private)`**
- Most variables are shared by default
  - A few exceptions: iteration varibales; stack variables in subroutines; automatic variables within a statement block.

Private Variables

- Not initialized at the beginning of parallel region.
- After parallel region
  - Not defined in OpenMP 2.x
  - 0 in OpenMP 3.x

tmp not initialized here

```
void wrong()
{
  int tmp=0;
  #pragma omp for private ( tmp )
        for(int j=0; j<100; ++j)
            tmp += j
  printf("%d\n", tmp )
}
```

OpenMP 2.5: tmp undefined

OpenMP 3.0: tmp is 0

Exercise 2: Calculate pi by Numerical Integration

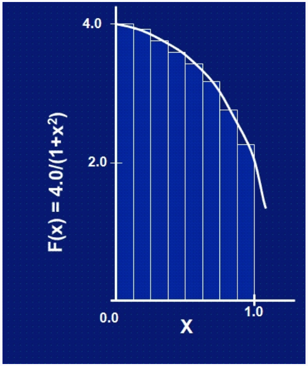
- We know that

$$\int_0^1 \frac{4.0}{(1+x)^2}\,dx = \pi$$

- So numerically, we can approximate pi as the sum of a number of rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Meadows et al, A “hands-on” introduction to OpenMP, SC09



Exercise 2: serial version

C/C++

```
#include <math.h>
#include <omp.h>
#include <stdio.h>
int main() {
  int N=1000000;
  double x,y,d;
  double pi,r=1.0;
  int i,sum=0;
  for (i=0;i<N;i++) {
    x = (double)(rand())/((double)
        (RAND_MAX)+(double)(1));
    y = (double)(rand())/((double)
        (RAND_MAX)+(double)(1));
    d = pow(2.*r*x-r,2)+pow(2.*r*y-r,2);
    if (d<pow(r,2)) sum++;
  }
  pi = 4.*(double)(sum)/(double)(N);
  printf("The value of pi is %f\n",pi);
}
```

Fortran

```
program pi_omp
  implicit none
  integer,parameter :: n=1000000
  real*8,parameter :: r=1.0
  integer i,sum
  real*8 x,y,d,pi
  sum=0
  do i=1,n
     call random_number(x)
     call random_number(y)
     d=(2*x*r-r)**2+(2*y*r-r)**2
     if (d.lt.r**2) sum=sum+1
  enddo
  pi=4*float(sum)/float(n)
  print *,'The value of pi is',pi
end program pi_omp
```

Exercise 2: OpenMP version

- Create a parallel version of the program with OpenMP

Special Cases of Private

- Firstprivate
  - Initialize each private copy with the corresponding value from the master thread
- Lastprivate
  - Allows the value of a private variable to be passed to the shared variable outside the parallel region

tmp initialized as 0

```
void wrong()
{
  int tmp=0;
  #pragma omp for firstprivate ( tmp ) lastprivate(tmp)
        for(int j=0; j<100; ++j)
            tmp += j
  printf("%d\n”, tmp )
}
```

The value of tmp is the value when j=99

Reduction

- The `reduction` clause allows accumulative operations on the value of variables.
- Syntax: `reduction (operator:variable list)`
- A private copy of each variable which appears in `reduction` is created as if the `private` clause is specified.
- Operators
  1. Arithmetic
  2. Bitwise
  3. Logical

Example: Reduction

C/C++

```
#include <omp.h>
int main() {
   int i,N=100,sum,a[100],b[100];
   for(i=0;i<N;++i){
      a[i]=i;
      b[i]=1;
   }
   sum = 0;
   #pragma omp parallel for
      reduction(+:sum)
      for(i=0;i<N;i++){
         sum=sum+a[i]*b[i];
      }
}
```

Fortran

```
program reduction
    implicit none
    integer i,n,sum,a(100),b(100)
    n= 100
    do i=1,n
       a(i) = i
    end do
    b = 1
    sum = 0
   $omp parallel do reduction(+:sum)
    do i=1,n
       sum = sum + a(i)*b(i)
    end do
end program reduction
```

Exercise 3: pi calculation with reduction

- Redo exercise 2 with reduction

Solution: pi calculation with reduction

C

```c
#include <omp.h>
#include <math.h>
#include <stdio.h>
int main() {
  int N=1000000;
  double x,y,d;
  double pi,r=1.0;
  int i,sum=0;
#pragma omp parallel for private(i,d,x,y) reduction(+:sum)
  for (i=0;i<N;i++) {
    x = (double)(rand())/((double)(RAND_MAX)+(double)(1));
    y = (double)(rand())/((double)(RAND_MAX)+(double)(1));
    d = pow(2.*r*x-r,2)+pow(2.*r*y-r,2);
    if (d<pow(r,2)) sum++;
  }
  pi = 4.*(double)(sum)/(double)(N);
  printf("The value of pi is %f\n",pi);
}
```

Fortran

```fortran
 program pi_omp
  implicit none
  integer,parameter :: n=1000000
  real*8,parameter :: r=1.0
  integer i,sum
  real*8 x,y,d,pi
  sum=0
  !$omp parallel do private(i,d,x,y) reduction(+:sum)
  do i=1,n
     call random_number(x)
     call random_number(y)
     d=(2*x*r-r)**2+(2*y*r-r)**2
     if (d.lt.r**2) sum=sum+1
  enddo
  !$omp end parallel do
  pi=4*float(sum)/float(n)
  print *,'The value of pi is',pi
end program pi_omp
```

Pitfalls: False Sharing

- Array elements that are in the same cache line can lead to false sharing.
  - The system handles cache coherence on a cache line basis, not on a byte or word basis.
  - Each update of a single element could invalidate the entire cache line.

```
!$omp parallel
myid=omp_get_thread_num()
nthreads=omp_get_num_threads()
do i=myid+1,n,nthreads
    a(i)=some_function(i)
end do
```

Pitfalls: Race Condition

- Multiple threads try to write to the same memory location at the same time.
  - Indeterministic results
- Inappropriate scope of varibale can cause indeterministic results too.
- When having indeterministic results, set the number of threads to 1 to check
  - If problem persists: scope problem
  - If problem is solved: race condition

```
!$omp parallel do
do i=1,n
    if (a(i) > max) then
      max = a(i)
    end if
end do
```

Synchronization: Barrier

- "Stop sign" where every thread waits until all threads arrive.
- Purpose: protect access to shared data.
- Syntax:
  - Fortran: `!$omp barrier`
  - C/C++: `#pragma omp barrier`
- A barrier is implied at the end of every parallel region
  - Use the `nowait` clause to turn it off
- Synchronizations are costly so their usage should be minimized.

Synchronization: Crtitical and Atomic

- Critical: Only one thread at a time can enter a `critical` region

```
!$omp parallel do
do i=1,N
    a = some_calculation(i)
    !$omp critical
    call some_function(a,x)
end do
!$omp end parallel do
```

- Atomic: Only one thread at a time can update a memory location

```
!$omp parallel do
do i=1,N
    b = some_calculation(i)
    !$omp atomic
    a = a + b
end do
!$omp end parallel do
```

Runtime Library Functions

- Modify/query the number of threads
  - `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()`, `omp_get_max_threads()`
- Query the number of processors
  - `omp_num_procs()`
- Query whether or not you are in an active parallel region
  - `omp_in_parallel()`
- Control the behavior of dynamic threads
  - `omp_set_dynamic()`, `omp_get_dynamic()`

Environment Variables

- OMP_NUM_THREADS: set default number of threads to use.
- OMP_SCHEDULE: control how iterations are scheduled for parallel loops.

References

- https://docs.loni.org/wiki/Using_OpenMP
- http://en.wikipedia.org/wiki/OpenMP
- http://www.nersc.gov/nusers/help/tutorials/openmp
- http://www.llnl.gov/computing/tutorials/openMP
- http://www.citutor.org