Introduction to OpenMP

Le Yan
Objectives of Training

- Acquaint users with the concept of shared memory parallelism
- Acquaint users with the basics of programming with OpenMP
Memory System: Shared Memory

- Global memory space, accessible by all processors
- Data sharing among processors achieved by reading/writing to the same memory location
- 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**
  - Memory
  - Bus
  - Processor

- **Distributed memory**
  - Network
  - Memory
Shared vs Distributed

**Distributed Memory**

- **Pros**
  - Memory scalable with number of processors
  - Easier and cheaper to build

- **Cons**
  - Difficult load balancing
  - Data sharing is slow

**Shared Memory**

- **Pros**
  - Global address space is user-friendly
  - Data sharing is fast

- **Cons**
  - Lack of scalability
  - Data conflict issues
OpenMP

- OpenMP has been the industry standard for shared memory programming over the last decade
  - Permanent members of the OpenMP Architecture Review Board: AMD, Cray, Fujitsu, HP, IBM, Intel, Microsoft, NEC, PGI, SGI, Sun
- OpenMP 3.0 was released in May 2008
- 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
Advantages of OpenMP

- Portability: standard among many shared memory platforms and hardware 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.
Hello World: C

```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;
}
```

**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
Hello World: Fortran

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
Building Blocks 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
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 processes
  - `omp_num_procs()`

- Query whether or not in an active parallel region
  - `omp_in_parallel()`

- Control the behavior of dynamic threads
  - `omp_set_dynamic()`, `omp_get_dynamic()`
Working 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 working sharing directives available
  - Sections
  - Tasks (new feature in OpenMP 3.0)
Example: Parallel Loops

**C/C++**

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

**Fortran**

```fortran
program paralleルドo
    implicit none
    integer i,n,a(100)
    i=0
    n=100
    !$omp parallel do private(i)
    do i=1,n
        a(i)=user_function(i)
    enddo
end program paralleルドo
```
Scope of Variables

- **Shared(list)**: specify the variables that are shared among all the threads

- **Private(list)**: create a local copy of the specified variables for each thread (the value is uninitialized!)

- **Default(shared|private|none)**: define the default scope of variables; C/C++ API does not have default(private)

- The “default” is **default(shared)**

- It is good programming practice to use **default(none)** and define variables explicitly
Load Balancing (1)

- OpenMP provides different methods to map loop iterations onto threads, indicated by the `schedule` clause.
- **Syntax:** `schedule(<method>, [chunk size])`
- **Methods include**
  - **Static:** the default schedule; divide iterations 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 the completion of the current one, until all iterations executed;
  - **Guided:** similar to dynamic; the only difference is that the chunk size starts large and shrinks to size eventually.
## Load Balancing (2)

### 4 threads, 100 iterations

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Iterations mapped onto thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Static</td>
<td>1-25</td>
</tr>
<tr>
<td>Static, 20</td>
<td>1-20, 81-100</td>
</tr>
<tr>
<td>Dynamic</td>
<td>1...</td>
</tr>
<tr>
<td>Dynamic, 10</td>
<td>1-10...</td>
</tr>
</tbody>
</table>
## Load Balancing (3)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Even and predictable workload per iteration; scheduling may be done at compilation time, least work at runtime;</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Highly variable and unpredictable workload per iteration; most work at runtime</td>
</tr>
<tr>
<td>Guided</td>
<td>Special case of dynamic scheduling; compromise between load balancing and scheduling overhead at runtime</td>
</tr>
</tbody>
</table>
Reduction

- The reduction clause allows accumulative operations on the value of variables
- Syntax: reduction(operator:variable list)
- A private copy of each variable appears in reduction is created as if the private clause is specified
- Operators
  - Arithmetic
  - Bitwise
  - 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
        b(i)=1
        sum=0
        !$omp parallel do reduction(+:sum)
        do i=1,n
            sum=sum+a(i)*b(i)
        enddo
        enddo
    end
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 if off
- Synchronizations are costly so their usage should be minimized (very important!)
Compilation and Execution

- **IBM p575 clusters**
  - Use the thread-safe compilers (with “_r”)
  - Use '-qsmp=omp' option

  %xlcr_r -qsmp=omp test.c && OMP_NUM_THREADS=4 ./a.out

- **Dell Linux clusters**
  - Use '-openmp' option (Intel compiler)

  %icc -openmp test.c && OMP_NUM_THREADS=4 ./a.out
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://www.nersc.gov/nusers/help/tutorials/openmp/
- http://www.llnl.gov/computing/tutorials/openMP/
Exercise 1: Hello World

- Write a “hello world” program with OpenMP where
  - If the thread id is odd, then print a message “Hello world from thread x, I'm odd!”
  - If the thread id is even, then print a message “Hello world from thread x, I am even!”
Solution

**C/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**

```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
```
Exercise 2: Calculate pi with Random Numbers

- Suppose we are throwing darts onto the board on the left
- The probability of falling in the circle is \( \frac{\pi}{4} \)
- We can calculate \( \pi \) by generating random points on the board and counting the fraction that falls in the circle
Exercise 2: Calculate pi with Random Numbers

- You can either start from scratch or parallelize the serial program pi.f90 or pi.c (under ~lyan1/traininglab/openmp)
- Set the number of darts large enough to avoid inaccurate result
- Random number generator (between 0 and 1)
  - C/C++: \( x = \text{double}(\text{rand()})/((\text{double})(\text{RAND\_MAX})+(\text{double})(1)) \)
  - Fortran: call random_number(x)
Solution: 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,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);
}
```
Solution: 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
  pi=4*float(sum)/float(n)
  print *,'The value of pi is',pi
end program pi_omp
```