OpenMP programming

Thomas Hauser
Director Research Computing
thomas.hauser@colorado.edu
Outline

- OpenMP
- Shared-memory model
- Parallel *for* loops
- Declaring private variables
- Critical sections
- Reductions
Parallelism at All Levels

• Parallelism across multiple nodes or process - MPI
• Parallelism across threads - OpenMP
• Parallelism across instructions
• Parallelism on data – SIMD Single Instruction Multiple Data
OpenMP

- OpenMP: An application programming interface (API) for parallel programming on multiprocessors
  - Compiler directives
  - Library of support functions
  - Environment variables
- OpenMP works in conjunction with Fortran, C, or C++
- [http://www.openmp.org](http://www.openmp.org)
Shared-memory Model

Processors interact and synchronize with each other through shared variables.
Fork/Join Parallelism

- Initially only master thread is active
- Master thread executes sequential code
- Fork: Master thread creates or awakens additional threads to execute parallel code
- Join: At end of parallel code created threads die or are suspended
Incremental Parallelization

- Sequential program a special case of a shared-memory parallel program
- Parallel shared-memory programs may only have a single parallel loop
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit at a time
Parallel for Loops

• Fortran programs often express data-parallel operations as do loops

\[
\text{do } i=1,nx, \text{ bs} \\
\quad a(i) = b(i) + c(i) \\
\text{enddo}
\]

• OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel
• Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads
Compiler directive

- Compiler directive in Fortran
- Compiler free to ignore directives
- Syntax:
  
  \[
  \text{!$OMP <rest of directive>} \\
  \text{!$OMP& continuation line} \\
  \text{!$ INTEGER :: i !something only in the parallel version of the program}
  \]

Include file `omp_lib.h` or module `omp_lib` for fortran
Include file `omp.h` for C
Pragma in C or C++

• Syntax:
  
  ```c
  #pragma omp <rest of directive>
  ```

  ```c
  #ifndef _OPENMP
  
  nt = omp_get_num_threads();
  ```

  ```c
  #endif
  ```
Parallel do directive

- Fortran:
  ```fortran
  !$OMP parallel do
do i=1,10
    a(i) = b(i) + c(i)
end do
  !$OMP end parallel do
  ```

- C
  ```c
  #pragma omp for
  for (i=0;i<10;i++){
    a[i] = b[i] + c[i];
  }
  ```

- Compiler must be able to verify the run-time system will have information it needs to schedule loop iterations
Execution Context

• Every thread has its own execution context
• Execution context: address space containing all of the variables a thread may access
• Contents of execution context:
  • static variables
  • dynamically allocated data structures in the heap
  • variables on the run-time stack
  • additional run-time stack for functions invoked by the thread
Shared and Private Variables

int main (int argc, char *argv[]) {
    int b[3];
    char *cptr;
    int i;

    cptr = malloc(1);
    #pragma omp parallel for
    for (i = 0; i < 3; i++)
        b[i] = i;
}

Heap
Stack

Master Thread (Thread 0)
Thread 1
Shared and Private Variables

- **Shared variable**: has same address in execution context of every thread
- **Private variable**: has different address in execution context of every thread
- A thread cannot access the private variables of another thread

- Clause specifies the scope of variables
  - Default: shared
  - `#pragma omp parallel private (var1, tmp)`
Declaring Private Variables

\[
\text{do } j = 1, nj \\
\quad \text{do } i = 1, ni \\
\quad \quad a(i,j) = \min(a(i,j), a(i,j)+\text{tmp}) \\
\quad \text{end do} \\
\text{end do}
\]

• Either loop could be executed in parallel
• We prefer to make outer loop parallel, to reduce number of forks/joins
• We then must give each thread its own private copy of variable \( i \)
Example Use of private Clause

```c
 !$OMP parallel do private(i)
 DO j = 1, nj
   DO i = 1, ni
     a(i,j) = MIN(a(i,j),a(i,k)+tmp)
   END DO
 END DO

 #pragma omp for private(i)
 for (j=0; j<nj; j++)
   for (i=0; i<ni; i++) {
     a[j,i] = a[j,i]+tmp
   }
```
Using the GCC compilers with OpenMP

- gfortran -fopenmp -g basic.f90
- gcc -fopenmp -g basic.c
- setenv OMP_NUM_THREADS 12 if tcsh/csh
- export OMP_NUM_THREADS=12 if bash/sh
- ./a.out
Using the Intel compilers with OpenMP

- ifort -openmp -g basic.f90
- icc -openmp -g basic.c
- setenv OMP_NUM_THREADS 12 if tcsh/csh
- export OMP_NUM_THREADS=12 if bash/sh
- ./a.out
parallel directive

• The `parallel` directive precedes a block of code that should be executed by *all* of the threads
• Note: execution is replicated among all threads
single Directive

• Suppose we only want to see the output once
• The single directive directs compiler that only a single thread should execute the block of code the directive precedes
• Syntax:

    !$omp single
Use of single Pragma

 !$omp parallel private(i,j) 
 Do i = 1, m 
  low = a(i) 
  high = b(i) 
  if (low > high) then 
  !$OMP single 
  write(*,*) "Exiting", I 
  !$OMP end single 
  exit 
 endif 
 End do 
 !$omp do 
 do j = low, high 
   c(j) = (c(j) - a(i))/b(i) 
 End do 
 !$omp end do 
 !$omp end parallel
Master directive

- Code is executed by the master thread
- Otherwise very similar to the single directive
Stream example

• Python notebook demo
Function omp_get_num_procs

• Returns number of physical processors available for use by the parallel program
• C: `int omp_get_num_procs(void);`
• Fortran:
  interface
    function omp_get_num_procs ()
      use omp_lib_kinds
      integer ( kind=omp_integer_kind ) :: omp_get_num_procs
    end function omp_get_num_procs
  end interface
Function omp_get_num_threads

• Function omp_get_num_threads returns the number of active threads
• If call this function from sequential portion of program, it will return 1

integer omp_get_num_threads ()
Function `omp_get_thread_num`

- This function returns the thread identification number
- If there are $t$ threads, the ID numbers range from 0 to $t-1$
- The master thread has ID number 0

```plaintext
integer omp_get_thread_num ()
```
Function `omp_set_num_threads`

- Uses the parameter value to set the number of threads to be active in parallel sections of code
- May be called at multiple points in a program

```c
subroutine omp_set_num_threads (t)
  integer, intent(in) :: t

void omp_set_num_threads(int num_threads);
```
Hello World - OpenMP

• See ipython notebook
firstprivate Clause

- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
- Variables are initialized once per thread, not once per loop iteration
- If a thread modifies a variable’s value in an iteration, subsequent iterations will get the modified value
Data Scoping

- Use:
  - `#pragma omp parallel default (none)`
- Initialization of privatized variable is required
  - `FIRSTPRIVATE ( var )` sets each private copy to previous global value
- Value of last iteration is needed
  - `LASTPRIVATE ( var )`
- Global variable needs to be private:
  - `THREADPRIVATE / COPYIN`
Use of firstprivate

program wrong
I = 10
!$OMP PARALLEL PRIVATE(I)
I = I + 1
print *, I
!$OMP END PARALLEL

program correct
I = 10
!$OMP PARALLEL FIRSTPRIVATE(I)
I = I + 1
print *, I
!$OMP END PARALLEL
lastprivate Clause

- Sequentially last iteration: iteration that occurs last when the loop is executed sequentially
- lastprivate clause: used to copy back to the master thread’s copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration
Numerical Integration

\[ \pi = \int_0^1 \frac{4.0}{1+x^2} \, dx \]

\[ \sum_{i=0}^{N} f(x_i) \, dx_i \approx \pi \]
Critical Sections

integer n, i
double precision w, x, area, pi, f, a

! function to integrate
f(a) = 4.d0 / (1.d0 + a*a)

w = 1.0d0/n
area = 0.0d0

do i = 1, n
        x = w * (i - 0.5d0)
        area = area + f(x)
end do
pi = w * area
Race Condition (cont.)

- If we simply parallelize the loop...

```fortran
area = 0.0d0

!$OMP PARALLEL DO PRIVATE(x)
do i = 1, n
    x = w * (i - 0.5d0)
    area = area + f(x)
end do
!$OMP END PARALLEL DO
pi = w * area
```
Race Condition (cont.)

• ... we set up a race condition in which one process may “race ahead” of another and not see its change to shared variable area

\[
\text{area} = \text{area} + \frac{4.0}{1.0 + x^2}
\]

Thread A

\[
15.432
\]

Thread B

\[
15.230
\]

Answer should be 18.995
Race Condition Time Line

Value of area

Thread A

Thread B

11.667

+ 3.765

11.667

+ 3.563

15.432

15.230

CU meetup 37 10/2/14
critical directive

• Critical section: a portion of code that only thread at a time may execute
• We denote a critical section by putting the directive

!$OMP critical

in front of a block of Fortran code
Correct, But Inefficient, Code

! function to integrate
f(a) = 4.d0 / (1.d0 + a*a)

w = 1.0d0/n
area = 0.0d0
!$OMP PARALLEL DO PRIVATE(X), SHARED(w,n)
do i = 1, n
    x = w * (i - 0.5d0)
    !$OMP CRITICAL
    area = area + f(x)
    !$OMP END CRITICAL
end do
!$OMP END PARALLEL DO
pi = w * area
Source of Inefficiency

- Update to area inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code
- Time to execute statement significant part of loop
- By Amdahl’s Law speedup will be severely constrained
Reductions

• Reductions are so common that OpenMP provides support for them
• May add reduction clause to \texttt{parallel do} directive
• Specify reduction operation and reduction variable
• OpenMP takes care of storing partial results in private variables and combining partial results after the loop
reduction Clause

• The reduction clause has this syntax: `reduction (<op> : <variable>)`
• Operators
  • +
  • *
  • -
  • .AND.
  • .OR.
  • .EQV.
  • .NEQV.
  • MAX
  • MIN
  • IAND
  • IOR
  • IEOR
π-finding Code with Reduction Clause

! function to integrate
f(a) = 4.d0 / (1.d0 + a*a)

w = 1.0d0/n
area = 0.0d0
!$OMP PARALLEL DO PRIVATE(X), SHARED(w,n)
!$OMP& REDUCTION(+:area)
do i = 1, n
    x = w * (i - 0.5d0)
    area = area + f(x)
end do
!$OMP END PARALLEL DO
pi = w * area
More General Data Parallelism

- Our focus has been on the parallelization of \texttt{do} loops
- Other opportunities for data parallelism
  - processing items on a “to do” list
  - \texttt{Do} loop + additional code outside of loop
Processing a “To Do” List

Diagram showing a process involving a heap, shared variables, master thread, task pointers (task_ptr), and thread 1.
Sequential Code (1/2)

Program taskQueue

Use taskList, only: taskIndex, process_task, get_next_task

Integer myIndex

myIndex = get_next_task()
do while (myindex /= -1)
    call process_task(myIndex)
    myIndex = get_next_task()
Enddo

end program taskQueue
Sequential Code (2/2)

Module taskQueue

Implicit none

Integer :: taskIndex

contains
  Integer function get_next_task()

  ! Check if we are out of tasks
  If (taskIndex == MAX_TASK) then
    get_next_task = -1
  Else
    taskIndex = taskIndex + 1
    get_next_task = taskIndex
  Return
  end function get_next_task
End module taskQueue
Parallelization Strategy

• Every thread should repeatedly take next task from list and complete it, until there are no more tasks
• We must ensure no two threads take same task from the list; i.e., must declare a critical section
Use of parallel directive

```c
!$OMP PARALLEL private(myIndex)
myIndex = get_next_task()
do while (myindex /= -1)
    call process_task(myIndex)
    myIndex = get_next_task()
Enddo

!$OMP END PARALLEL
```
Critical Section for `get_next_task`

Integer function `get_next_task()`

```plaintext
!$OMP critical
  ! Check if we are out of tasks
  If (index == MAX_TASK) then
    get_next_task == -1
  Else
    taskIndex = taskIndex + 1
    get_next_task = taskIndex
  Return
end function get_next_task
!$OMP end critical
```
Functional Parallelism

• To this point all of our focus has been on exploiting data parallelism
• OpenMP allows us to assign different threads to different portions of code (functional parallelism)
Functional Parallelism Example

\begin{align*}
v &= \text{alpha}() \\
w &= \text{beta}() \\
x &= \text{gamma}(v, w) \\
y &= \text{delta}() \\
\text{write(*,*}) &= \text{epsilon}(x,y)
\end{align*}

May execute alpha, beta, and delta in parallel
parallel sections Directive

• Precedes a block of \( k \) blocks of code that may be executed concurrently by \( k \) threads

• Syntax:

\[
!\text{omp parallel sections}
\]
section directive

- Precedes each block of code within the encompassing block preceded by the parallel sections directive
- May be omitted for first parallel section after the parallel sections directive
- Syntax:

  `!$omp section`
Example of \textit{parallel sections}

\begin{verbatim}
!$omp parallel sections
!$omp section
  v = alpha()
!$omp section
  w = beta()
!$omp section
  y = delta
!$omp end parallel sections
  x = gamma(v, w)
write(*,*) epsilon(x,y)
\end{verbatim}
Another Approach

Execute alpha and beta in parallel. Execute gamma and delta in parallel.
sections directive

• Appears inside a parallel block of code
• Has same meaning as the parallel sections pragma
• If multiple sections pragmas inside one parallel block, may reduce fork/join costs
Use of sections Directive

$\!$omp parallel
  $\!$omp sections
    $v = \alpha()$
  $\!$omp section
    $w = \beta()$
  $\!$omp end sections
$\!$omp sections
  $x = \gamma(v, w)$
  $\!$omp section
  $y = \delta()$
  $\!$omp end sections
write(*,*) $\epsilon(x, y)$
$\!$omp end parallel
Summary

• OpenMP an API for shared-memory parallel programming
• Shared-memory model based on fork/join parallelism
• Data parallelism