Shared Memory Parallel Programming

OpenMP

Environment and Synchronization
So Far…

• Parallel region
  #pragma omp parallel
  !$OMP PARALLEL

• Work sharing constructs
  !$OMP DO
  !$OMP WORKSHARE
  #pragma omp for
  #pragma omp sections
  #pragma omp single
Reminder

• Other features
  conditional parallelism
  loop scheduling
Data Environment Directives

• All variables are by default shared
  – all threads access the same memory location

• Exception: the loop variable of a parallel for/parallel do is private

• By using data directives, some variables can be made private to a thread
  – each thread has its own copy, memory location
Matrix Multiply

```c
#pragma omp parallel for
for( i=0; i<n; i++ )
    for( j=0; j<n; j++ ) {
        c[i][j] = 0.0;
        for( k=0; k<n; k++ )
            c[i][j] += a[i][k]*b[k][j];
    }
```

• a, b, c are shared
• i, j, k are private
Private Variables

```c
#pragma omp parallel for private( list )
```

- Compiler sets up a private copy of each variable in the list for each thread
- Our examples use OpenMP for and DO
- But these apply to other region and work-sharing directives
- For compiler buffs: thread has its own stack
Private Variables: Example (1 of 2)

```c
for( i=0; i<n; i++ ) {
    tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
}
```

- Swaps the values of a and b
- Loop-carried dependence on tmp
- Easily fixed by privatizing tmp
Private Variables: Example (2 of 2)

```c
#pragma omp parallel for private ( tmp )
for( i=0; i<n; i++ ) {
    tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
}
```

- Removes dependence on tmp
- Would be more difficult to do in Pthreads
Private Variables: Alternative 1

\[
\text{for}\ (i=0; \ i<n; \ i++ ) \ \{ \\
\quad \text{tmp}[i] = a[i]; \\
\quad a[i] = b[i]; \\
\quad b[i] = \text{tmp}[i]; \\
\}
\]

• Requires sequential program change.
• Wasteful in space, \(O(n)\) vs. \(O(p)\).
#pragma omp parallel private (iam, …) for
for ( i = 0; i = numthreads; i++) {
    iam = omp_get_thread_num(); s= iam / …  
    f(s);
}
f(from)
{
    int tmp; /* local allocation on stack */
    for( i=from; i<from+const; i++ ) {
        tmp = a[i];         
        a[i] = b[i];         
        b[i] = tmp;
    }
}
Remember Example

!$OMP PARALLEL PRIVATE (iam, np, ipoints)
  iam = omp_get_thread_num( )
  np   = omp_get_num_threads ( )
  ipoints = npoints / np
  call domywork ( x, iam, ipoints)
!$OMP END PARALLEL
Default Behavior

```c
!$OMP PARALLEL DEFAULT (PRIVATE) SHARED (X, NPOINTS )
iam = omp_get_thread_num( )
np  = omp_get_num_threads ( )
ipoints = npoints / np
    call domywork ( x, iam, ipoints)
!$OMP END PARALLEL
```

- We can set the default to be either private or shared
Firstprivate and Lastprivate

- The initial and final values of private variables are unspecified.
- A firstprivate variable is private, and the private copies are initialized using its value before the loop.
- A lastprivate variable is private, and the thread executing the {sequentially last iteration/lexically last section} updates the version of the object outside the parallel region.
Firstprivate/Lastprivate: Example (1 of 2)

```c
for(i=0; (i<n) && b[i]; i++ )
    a[i] = b[i];
for(j=i; j<n; j++ )
    a[j] = 1.0;
```

- Sets all elements of \( a \) to the value of the corresponding element in \( b \), up to first zero value in \( b \)
- Sets all further elements of \( a \) to 1.0
Firstprivate/Lastprivate: Example

```c
#pragma omp parallel for lastprivate(i)
for(i=0; (i<n) && b[i]; i++)
    a[i] = b[i];
#pragma omp parallel for firstprivate(i)
for(j=i; j<n; j++)
    a[j] = 1.0;
```
Firstprivate/Lastprivate: Example

!$OMP PARALLEL
!$OMP DO LASTPRIVATE ( i )
   DO i = 1, n
      a ( i ) = b ( i ) + c ( i )
   END DO
!$OMP END PARALLEL
CALL NEXT ( i )
Threadprivate

- Variables are private within an entire parallel region
- Threadprivate variables are global variables that are private throughout the execution of the program
Threadprivate

#pragma omp threadprivate( list )
Example: #pragma omp threadprivate( x)

- Requires program change in Pthreads
- Requires an array of size p
- Access as x[thread_self()]
- Costly if accessed frequently
- Not cheap in OpenMP either
Reduction Variables: Example

```
#pragma omp parallel for reduction( +:sum )
for( i=0; i<n; i++ )
    sum += a[i];
```

- Sum is automatically initialized to zero
Reduction Variables: Example

```c
!$OMP PARALLEL DO reduction( max: m )
do i = 1, 100
    m  = max ( m, sin ( real ( i ) ) )
end do
```

- m is automatically initialized to smallest integer
Reduction Variables

#pragma omp parallel for reduction(op:list)

• op is one of +, *, -, &, ^, |, &&, or ||
• The variables in list must be used with this operator in the loop
• A private copy is created for each thread
• The copies are automatically initialized to sensible values
Reduction Variables

!$OMP PARALLEL DO reduction(op:list)

• op is one of +, *, -, .AND., .OR., .EQV., .NEQV.

• New Fortran 2.0 standard extends this to: MAX, MIN, IAND, IOR, IEOR

• Permits reductions on array elements (very common in scientific applications)
OpenMP Jacobi Code with Convergence

```c
for( ; diff > delta; ) {
    #pragma omp parallel for
    for (i=0; i<n; i++)
        for( j=0; j<n, j++ ) { … } 
    diff = 0;
    #pragma omp parallel for reduction( max: diff )
    for( i=0; i<n; i++)
        for( j=0; j<n; j++ ) {
            diff = max(diff, fabs(grid[i][j] - temp[i][j]));
            grid[i][j] = temp[i][j];
        }
}
```

No reduction operator for max or min in C
Data Environment Directives: Summary

• For good performance, OpenMP code should use private variables wherever possible
  – reduces cache problems
• However, this could waste a lot of memory
• Use of reductions also extremely important
• New version of standard fixes major practical problems with reductions for Fortran (90)
Summary: Data Environment Directives

- Private
- Firstprivate
- Lastprivate
- Reduction
- Threadprivate
- Copyin
Synchronization Primitives

• Critical
  
  #pragma omp critical name

  Implements critical sections by name.

  Similar to Pthreads mutex locks (name ~ lock).

• Barrier
  
  #pragma omp barrier

  Implements global barrier.
OpenMP Jacobi with Convergence (1 of 2)

```c
#pragma omp parallel private( mydiff )
for( ; diff > delta; ) {
    #pragma omp for nowait
    for( i=from; i<to; i++ )
        for( j=0; j<n, j++ ) { … }  
    diff = 0.0;
    mydiff = 0.0;
    #pragma omp barrier
...
```
... 
#pragma omp for nowait
for( i=from; i<to; i++ )
    for( j=0; j<n; j++ ) {
        mydiff=max(mydiff,fabs(grid[i][j]-temp[i][j]));
        grid[i][j] = temp[i][j];
    }
#pragma critical
    diff = max( diff, mydiff );
#pragma barrier
}
Synchronization Primitives

• No condition variables
• Result: must busy wait for condition synchronization
• Clumsy
• Very inefficient on some architectures
PIPE: Sequential Program

for( i=0; i<num_pic, read(in_pic); i++ ) {
    int_pic_1 = trans1( in_pic );
    int_pic_2 = trans2( int_pic_1);
    int_pic_3 = trans3( int_pic_2);
    out_pic = trans4( int_pic_3);
}
PIPE: Parallel Program

P0: for( i=0; i<num_pics; i++ ) {
    int_pic_1[i] = trans1( in_pic );
    signal(event_1_2[i]);
}
P1: for( i=0; i<num_pics; i++ ) {
    wait( event_1_2[i] );
    int_pic_2[i] = trans2( int_pic_1[i] );
    signal(event_2_3[i]);
}
PIPE: Main Program

#pragma omp parallel sections
{
    #pragma omp section
    stage1();
    #pragma omp section
    stage2();
    #pragma omp section
    stage3();
    #pragma omp section
    stage4();
}
PIPE: Stage 1

void stage1()
{
    num1 = 0;
    for( i=0; i<num_pics, read(in_pic); i++ ) {
        int_pic_1[i] = trans1( in_pic );
        #pragma omp critical 1
        num1++;
    }
}
void stage2 ()
{
    for( i=0; i<num_pic; i++ ) {
        do {
            #pragma omp critical 1
            cond = (num1 <= i);
        } while (cond);
        int_pic_2[i] = trans2(int_pic_1[i]);
        #pragma omp critical 2
        num2++;
    }
}
OpenMP PIPE

• Note the need to exit critical during wait
• Otherwise no access by other thread
• Never busy-wait inside critical!
Sequential TSP: Code Outline

init_q(); init_best();
while( (p=de_queue()) != NULL ) {
    for each expansion by one city {
        q = add_city(p);
        if( complete(q) ) { update_best(q) };
        else { en_queue(q) };
    }
}
OpenMP TSP

• An exercise
• Be careful:
  – Cannot simply use critical in de_queue(), en_queue()
  – If thread busy-waits inside critical, no progress!
• Careful use of critical on parts of en_queue(), de_queue()
OpenMP Synchronization

- Barrier
- Ordered Section
- Atomic update
- Locks
Difficult Stuff

• Flush
  – how do you know that this is needed?
• Dynamic vs. static threads
• Role of environment
  – esp. other user codes on ccNUMA
• Nesting
  – not implemented yet in general
OpenMP Summary

• Directives for shared memory parallel programming
• Fortran and C/C++
• Compiler translates directives
• But user can have big impact on performance via careful use of features
• Much easier to use than MPI
OpenMP Summary

• Initially targeted engineering applications
• But more widely applicable than that
• Many other features proposed
  – e.g. support for task queues
  – The ARB is working on defining this and other extensions
• OpenMP may need to be suitable for big variety of programs on multicore multithreading platforms
OpenMP Performance

• Performance of different constructs depends on compiler and target machine
• EPCC low-level benchmarks
• LLNL benchmarks
OpenMP Applicability

- Can express fine and coarse-grain parallelism
- Can be compiled for SMPs and DSM systems
- Can be combined with MPI for cluster of SMPs
- Software DSM can compile it for SMP clusters
OpenMP: Limits

• OpenMP shares program’s work among threads
  – It says nothing about how to store data that is not private
  – Because assumption is that memory is shared and cost of access is same for all processors (threads)

• This is not true for non-uniform memory access (NUMA) and distributed memory systems
  – NUMA machines are popular
  – vendor extensions enable OpenMP on ccNUMA (DSM) platforms