6: Using OpenMP

http://people.sc.fsu.edu/~jburkardt/presentations/fdi_2008_lecture6.pdf

.....

John Burkardt
Information Technology Department
Virginia Tech

.....

FDI Summer Track V:
Parallel Programming

10-12 June 2008

Using OpenMP

It's time to take a closer look at how **OpenMP** works.

- OpenMP Environment Variables and Functions
- Parallel Control Structures
- SATISFY: Parallel Computing Without Loops
- Data Classification (Shared/Private)
- Data Dependency

OpenMP Environment

OpenMP uses internal data which can be of use or interest.

In a few cases, the user can set some of these values by means of a Unix environmental variable.

There are also functions the user may call to get or set this information.

OpenMP Environment

You can **set**:

- maximum number of threads - most useful!
- details of how to handle loops, nesting, and so on

You can **get**:

- number of "processors" (=cores) are available
- individual thread id's
- maximum number of threads
- wall clock time

OpenMP Environment: Variables

If you are working on a UNIX system, you can "talk" to **OpenMP** by setting certain environment variables.

The syntax for setting such variables varies slightly, depending on the shell you are using.

Many people use this method in order to specify the number of threads to be used. If you don't set this variable, your program runs on one thread.

OpenMP Environment: Variables

There are just 4 **OpenMP** environment variables:

- **OMP_NUM_THREADS**, maximum number of threads
- **OMP_DYNAMIC**, allows dynamic thread adjustment
- **OMP_NESTED**, allows nested parallelism, default 0/FALSE
- **OMP_SCHEDULE**, determines how loop work is divided up

OpenMP Environment: Variables

Determine your shell by:

echo $SHELL

Set the number of threads in the Bourne, Korn and Bash shells:

export OMP_NUM_THREADS=4

In the C or T shells, use a command like

setenv OMP_NUM_THREADS 4

To verify:

echo $OMP_NUM_THREADS

OpenMP Environment: Functions

OpenMP environment functions include:

- **omp_set_num_threads (t_num)**
- **t_num = omp_get_num_threads ( )**
- **p_num = omp_get_num_procs ( )**
- **t_id = omp_get_thread_num ( )**
- **wtime = omp_get_wtime()**

OpenMP Environment: How Many Threads May I Use?

A **thread** is one of the "workers" that OpenMP assigns to help do your work.

There is a limit of

- 1 thread in the sequential sections.
- **T_NUM** threads in the parallel sections.

OpenMP Environment: How Many Threads May I Use?

T_NUM

- has a default for your computer.
- can be initialized by setting **OMP_NUM_THREADS** before execution
- can be reset by calling **omp_set_num_threads(t_num)**
- can be checked by calling **t_num=omp_get_num_threads()**

OpenMP Environment: How Many Threads Should I Use?

If **T_NUM** is 1, then you get no parallel speed up at all, and probably actually slow down.

You can set **T_NUM** much higher than the number of processors; some threads will then "share" a processor.

Reasonable: one thread per processor.

```
p_num = omp_get_num_procs ( );
t_num = p_num;
omp_set_num_threads ( t_num );
```

These three commands can be compressed into one.

OpenMP Environment: Which Thread Am I Using?

In any parallel section, you can ask each thread to identify itself, and assign it tasks based on its index.

```
!$omp parallel
  t_id = omp_get_thread_num ( )
  write ( *, * ) 'Thread ', t_id, ' is running.'
!$omp end parallel
```

OpenMP Environment: How Much Time Has Passed?

You can take "readings" of the wall clock time before and after a parallel computation.

```
    wtime = omp_get_wtime ( );
#pragma omp parallel for
    for ( i = 0; i < n; i++ )
    {
      Do a lot of work in parallel;
    }
    wtime = omp_get_wtime ( ) - wtime;

    cout << "Work took " << wtime << " seconds.\n";
```

OpenMP Environment: How Much Time Has Passed?

You can take "readings" of the wall clock time before and after a parallel computation.

```
    wtime = omp_get_wtime ( );
#pragma omp parallel for
    for ( i = 0; i < n; i++ )
    {
      Do a lot of work in parallel;
    }
    wtime = omp_get_wtime ( ) - wtime;

    cout << "Work took " << wtime << " seconds.\n";
```

"Hiding" Parallel Code

OpenMP tries to make it possible for you to have your sequential code and parallelize it too. In other words, a single program file should be able to be run sequentially or in parallel, simply by turning on the directives.

This isn't going to work so well if we have these calls to **omp_get_wtime** or **omp_get_proc_num** running around. They will cause an error when the program is compiled and loaded sequentially, because the **OpenMP** library will not be available.

Fortunately, you can "comment out" all such calls, just as you do the directives, or, in C and C++, check whether the symbol **_OPENMP** is defined.

Hiding Parallel Code in C++

```
# ifdef _OPENMP
  # include <omp.h>
# endif
# ifdef _OPENMP
   wtime = omp_get_wtime ( );
# endif
#pragma omp parallel for
   for ( i = 0; i < n; i++ ){
     Do a lot of work in parallel;  }
# ifdef _OPENMP
   wtime = omp_get_wtime ( ) - wtime;
   cout << "Work took " << wtime << " seconds.\n";
# else
   cout << "Elapsed time not measured.\n";
# endif
```

Hiding Parallel Code in FORTRAN90

```
!$omp use omp_lib

!$omp wtime = omp_get_wtime ( )
!$omp parallel do
  do i = 1, n
      Do a lot of work in parallel;
  end do
!$omp end parallel do
!$omp wtime = omp_get_wtime ( ) - wtime
!$omp write ( *, * ) 'Work took', wtime, ' seconds.'
```

Parallel Control Structures, Loops

```
#pragma omp parallel for
for ( i = ilo; i <= ihi; i++ )
{
  C/C++ code to be performed in parallel
}
```

```
!$omp parallel do
do i = ilo, ihi
  FORTRAN code to be performed in parallel
end do
!$omp end parallel do
```

Parallel Control Structure, Loops

FORTRAN Loop Restrictions:

The loop must be a **do** loop of the form;

```
do i = low, high (, increment)
```

The limits **low**, **high** (and **increment** if used), cannot change during the iteration.

The program cannot jump out of the loop, using an **exit** or **goto**.

The loop cannot be a **do while**, and it cannot be a **do** with no iteration limits.

Parallel Control Structure, Loops

C Loop Restrictions:

The loop must be a **for** loop of the form:

```
for ( i = low; i < high; increment )
```

The limits **low** and **high** cannot change during the iteration;

The **increment** (or decrement) must be by a fixed amount.

The program cannot **break** from the loop.

Parallel Control Structures, No Loop

It is possible to set up parallel work without a loop.

In this case, the user can assign work based on the ID of each thread.

For instance, if the computation models a crystallization process over time, then at each time step, half the threads might work on updating the solid part, half the liquid.

If the size of the solid region increases greatly, the proportion of threads assigned to it could be increased.

Parallel Control Stuctures, No Loop, C/C++

```
#pragma omp parallel
{
  t_id = omp_get_thread_num ( );
  if ( t_id % 2 == 0 )
  {
    solid_update ( );
  }
  else
  {
    liquid_update ( );
  }
}
```

Parallel Control Stuctures, No Loop, FORTRAN

```
!$omp parallel
  t_id = omp_get_thread_num ( )
  if ( mod ( t_id, 2 ) == 0 ) then
    call solid_update ( )
  else if ( mod ( t_id, 4 ) == 1 ) then
    call liquid_update ( )
  else if ( mod ( t_id, 4 ) == 3 ) then
    call gas_update ( )
  end if
!$omp end parallel
```

(Now we've added a gas update task as well.)

Parallel Control Structures, WORKSHARE

FORTRAN90 expresses implicit vector operations using colon notation.

OpenMP includes the **WORKSHARE** directive, which says that the marked code is to be performed in parallel.

The directive can also be used to parallelize the FORTRAN90 **WHERE** and the FORTRAN95 **FORALL** statements.

Parallel Control Stuctures, FORTRAN90

```
!$omp workshare
  y(1:n) = a * x(1:n) + y(1:n)
!$omp end workshare
```

```
!$omp workshare
  where ( x(1:n) /= 0.0 )
    y(1:n) = log ( x(1:n) )
  elsewhere
    y(1:n) = 0.0
  end where
!$omp end workshare
```

Parallel Control Stuctures, FORTRAN95

```
!$omp workshare
  forall ( i = k+1:n,j = k+1:n )
    a(i,j) = a(i,j) - a(i,k) * a(k,j)
  end forall
!$omp end workshare
```

(This calculation corresponds to one of the steps of Gauss elimination or LU factorization)

SATISFY: Parallel Computing Without Loops

OpenMP is easiest to use with loops.

Here is an example where we get parallel execution without using loops.

Doing the problem this way will make **OpenMP** seem like a small scale version of **MPI**.

SATISFY: Problem specification

What sets of 16 logical input values **X** will cause the following function to have the value **TRUE**?

```
f(x) = (  x(1)  ||  x(2)  ) && ( !x(2)  || !x(4)  ) &&
       (  x(3)  ||  x(4)  ) && ( !x(4)  || !x(5)  ) &&
       (  x(5)  || !x(6)  ) && (  x(6)  || !x(7)  ) &&
       (  x(6)  ||  x(7)  ) && (  x(7)  || !x(16) ) &&
       (  x(8)  || !x(9)  ) && ( !x(8)  || !x(14) ) &&
       (  x(9)  ||  x(10) ) && (  x(9)  || !x(10) ) &&
       ( !x(10) || !x(11) ) && (  x(10) ||  x(12) ) &&
       (  x(11) ||  x(12) ) && (  x(13) ||  x(14) ) &&
       (  x(14) || !x(15) ) && (  x(15) ||  x(16) )
```

SATISFY: Problem specification

Sadly, there is no clever way to solve a problem like this in general. You simply try every possible input.

How do we generate all the inputs?

Can we divide the work among multiple processors?

SATISFY: Algorithm Design

There are $2^{16} = 65,536$ distinct input vectors.

There is a natural correspondence between the input vectors and the integers from 0 to 65535.

We can divide the range [0,65536] into **T_NUM** distinct (probably unequal) subranges.

Each thread can generate its input vectors one at a time, evaluate the function, and print any successes.

SATISFY: Program Design

```
#pragma omp parallel
{
  T_NUM = omp_get_num_threads ( );
  T_ID = omp_get_thread_num ( );
  ILO = ( T_ID     ) * 65535 / T_NUM;
  IHI = ( T_ID + 1 ) * 65535 / T_NUM;

  for ( I = ILO; I < IHI; I++ )
  {
    X[0:15] <= I              (set binary input)
    VALUE = F ( X )           (evaluate function)
    if ( VALUE ) print X
  end
}
```

SATISFY: FORTRAN90 Implementation

```
  thread_num = omp_get_num_threads ( )
  solution_num = 0
!$omp parallel private ( i, ilo, ihi, j, value, x ) &
!$omp shared ( n, thread_num ) &
!$omp reduction ( + : solution_num )
  id = omp_get_thread_num ( )
  ilo = id * 65536 / thread_num
  ihi = ( id + 1 ) * 65536 / thread_num

  j = ilo
  do i = n, 1, -1
    x(i) = mod ( j, 2 )
    j = j / 2
  end do

  do i = ilo, ihi - 1
    value = circuit_value ( n, x )
    if ( value == 1 ) then
      solution_num = solution_num + 1
      write ( *, '(2x,i2,2x,i10,3x,16i2)' )  solution_num, i - 1, x(1:n)
    end if
    call bvec_next ( n, x )
  end do
!$omp end parallel
```

SATISFY: Observations

I wanted an example where parallelism didn't require a **for** or **do** loop. The loop you see is carried out entirely by one (each) thread.

The "implicit loop" occurs when when we begin the parallel section and we generate all the threads.

The idea to take from this example is that the environment functions allow you to set up your own parallel structures in cases where loops aren't appropriate.

Data Classification (Private/Shared)

The very name "shared memory" suggests that the threads share one set of data that they can all "touch".

By default, **OpenMP** assumes that all variables are to be shared – with the exception of the loop index in the **do** or **for** statement.

It's obvious why each thread will need its own copy of the loop index. Even a compiler can see that!

However, some other variables may need to be treated specially when running in parallel. In that case, you must explicitly tell the compiler to set these aside as **private** variables.

It's a good practice to explicitly declare all variables in a loop.

Data Classification (Private/Shared)

```
do i = 1, n
  do j = 1, n
    d = 0.0
    do k = 1, 3
      dif(k) = coord(k,i) - coord(k,j)
      d = d + dif(k) * dif(k)
    end do
    do k = 1, 3
      f(k,i) = f(k,i) - dif(k) * pfun ( d ) / d
    end do
  end do
end do
```

Data Classification (Private/Shared)

I've had to cut this example down a bit. So let me point out that **coord** and **f** are big arrays of spatial coordinates and forces.

The variable **n** is counting particles, and where you see a 3, that's because we're in 3-dimensional space.

The mysterious **pfun** is a function that evaluates a factor that will modify the force.

You should list all the variables that show up in this loop, and try to determine if they are **shared** or **private**.

Data Classification (Private/Shared)

```
!$omp parallel do private ( i, j, k, d, dif ) &
!$omp shared ( n, coord, f )
  do i = 1, n
    do j = 1, n
      d = 0.0
      do k = 1, 3
        dif(k) = coord(k,i) - coord(k,j)
        d = d + dif(k) * dif(k)
      end do
      do k = 1, 3
        f(k,i) = f(k,i) - dif(k) * pfun ( d ) / d
      end do
    end do
  end do
!$omp end parallel do
```

Data Classification (Private/Shared/Reduction)

We have already seen examples where a variable was used to collect a sum. We had to use the **reduction** clause to handle this case.

If a variable is declared in a **reduction** clause, it *does not* also get declared as private or shared!

(A "reduction" variable has some features of both shared and private variables.)

Data Classification (Private/Shared/Reduction)

Suppose in FORTRAN90 we need the maximum of a vector.

```
x_max = - huge ( x_max )                ---+
do i = 1, n                                |
  x_max = max ( x_max, x(i) )              |  Loop #1
end do                                  ---+

x_max = maxval ( x(1:n) )               --->  Loop #2
```

In loop #2, we give the compiler freedom to do the calculation the best it can. Is this always the solution? In an actual computation, we might only compute the vector X one element at a time, so we would never have an actual array to process.

Please suggest how we would parallelize loop #1 or loop #2!

Data Classification (Private/Shared/Reduction)

```
!$omp parallel do private ( i ) shared ( n, x ) &
!$omp reduction ( max : x_max )
do i = 1, n
  x_max = max ( x_max, x(i) )
end do
!$omp end parallel do

!$omp parallel workshare
  x_max = maxval ( x(1:n) )
!$omp end parallel workshare
```

Data Classification (Private/Shared/Reduction)

In Loop #1, could we also do this?

```
!$omp parallel do private ( i ) shared ( n, x ) &
!$omp reduction ( max : big )
do i = 1, n
  if ( big < x(i) ) then
    big = x(i)
  end if
end do
!$omp end parallel do
```

(Yes, but we should initialize **big** or the sequential code will be incorrect.)

Random Numbers and OpenMP

Random numbers are a vital part of many algorithms. But you must be sure that your random number generator behaves properly.

It is acceptable (but hard to check) that your parallel random numbers are at least "similarly distributed."

It would be ideal if you could generate the same stream of random numbers whether in sequential or parallel mode.

Random Numbers and OpenMP

Most random number generators work by repeatedly "scrambling" an integer value called the seed. One kind of scrambling is the linear congruential generator:

```
SEED = ( A * SEED + B ) modulo C
```

If you want a real number returned, this is computed indirectly, by an operation such as

```
R = ( double ) SEED / 2147483647.0
```

Most random number generators hide the seed internal in static memory, initialized to a default value, which you can see or change only by calling the appropriate routine.

Random Numbers and OpenMP

Some system random number generators will work properly under OpenMP, but it's very important to test them. Initialize the seed to 123456789 (for example), and compute 20 random values sequentially. Repeat the process in parallel and compare.

```
SEED = ( A * SEED + B ) modulo C
```

If you want a real number returned, this is computed indirectly, by an operation such as

```
R = ( double ) SEED / 2147483647.0
```

Most random number generators hide the seed internal in static memory, initialized to a default value, which you can see or change only by calling the appropriate routine.

Random Numbers and OpenMP

```
# include ...stuff...
int main ( void )
{
  int i;
  unsigned int seed = 123456789;
  double y[20];

  srand ( seed );
  for ( i = 0; i < 20; i++ )
  {
    y[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
  }
  return 0;
}
```

Random Numbers and OpenMP

Make a parallel version of this program and compare the results. But even if you happen to get the same results, I still am not comfortable with this!

If you can, you should seek a random number function whose seed is an explicit argument.

Secondly, it seems to me you can't in general, hope to set up a random number generator that allows you to compute the "50th" random number immediately, because of the way they are set up.

So perhaps a compromise is this: use a parallel section, set a seed based on the thread index, and then start a loop.

Random Numbers and OpenMP

```
#omp pragma parallel private ( i, id, r, seed )
  id = omp_get_thread_num ( );
  seed = 123456789 * id
  for ( i = 0; i < 1000; i++ )
  {
    r = my_random ( seed );
    (do stuff with random number r )
  }
#omp pragma end parallel
```

Random Numbers and OpenMP

Do you see why I have made my choices this way?

Do you see why I am still unhappy with this setup? (we're not really emulating a sequential version.) (when you pick several seed arbitrary, it's actually possible for one sequence to overlap another)

After setting SEED, could I call srand (seed) and then use the system **rand()** function?

Note that, for MPI, there is at least one package, called **SPRNG**, which can generate random numbers that are guaranteed to be well distributed.

Data Dependency - Adding Digits Base B

Suppose vectors X and Y contain digits base B, and that Z is to hold the base B representation of their sum. (Let's assume for discussion that base B is 10).

Adding is easy. But then we have to carry. Every entry of Z that is B or greater has to have the excess subtracted off and carried to the next higher digit. This works in one pass of the loop only if we start at the lowest digit.

And adding 1 to 9,999,999,999 shows that a single carry operation could end up changing every digit we have.

Data Dependency

```
do i = 1, n
  z(i) = x(i) + y(i)
end do
overflow = .false.
do i = 1, n
  carry = z(i) / b
  z(i) = z(i) - carry * b
  if ( i < n ) then
    z(i+1) = z(i+1) + carry
  else
    overflow = .true.
  end if
end do
```

Data Dependency - Adding Digits Base B

In the carry loop, notice that on the I-th iteration, we might write (modify) both **z[i]** and **z[i+1]**.

In parallel execution, the value of **z[i]** used by iteration I might be read as 17, then iteration I-1, which is also executing, might change the 17 to 18 because of a carry, but then iteration I, still working with its temporary copy, might carry the 10, and return the 7, meaning that the carry from iteration I-1 was lost!

99% of carries in base 10 only affect at most two higher digits. So if we were desperate to use parallel processing, we could use repeated carrying in a loop, plus a temporary array **z2**.

Data Dependency - Adding Digits Base B

```
  do
!$omp workshare
    z2(1)   = mod ( z(1)   , b )
    z2(2:n) = mod ( z(2:n), b ) + z(1:n-1) / b
    z(1:n)  = z2(1:n)
    done = all ( z(1:n-1) / b == 0 )
!$omp end workshare
    if ( done )
      exit
    end if

  end do
```

Of Things Not Said

Although OpenMP is a relatively simple programming system, there is a lot we have not covered.

Here is an example of a simple problem that OpenMP can handle using ideas I haven't discussed.

In Gauss elimination, you need to find the maximum entry in an array. Well, we know how to do that. But actually, you need the **index** of the maximum entry in the array. This is like a reduction variable, but the reduction clause doesn't include this category.

OpenMP can solve this problem by having a shared variable that has an associated "lock". The lock implies that, if a thread wants to modify the value, it has to request that it be given temporary access to the variable, during which other threads cannot change it.

Of Things Not Said

OpenMP can also mark a section of the loop as "critical" which means that only one thread at a time can be executing those instructions.

There are also ways to override the default rules for how the work in a loop is divided, to force threads to wait for an event, and other items to orchestrate your program's execution.

Debugging a parallel programming can be quite difficult. If you are familiar with the Berkeley **dbx** or Gnu **gdb** debuggers, these have been extended to deal with multithreaded programs. There is also a program called **TotalView** with an intuitive graphical interface.

Of Things Not Said

Good references include:

1. **Chandra**, *Parallel Programming in OpenMP*
2. **Chapman**, *Using OpenMP*
3. **Petersen, Arbenz**, *Introduction to Parallel Programming*
4. **Quinn**, *Parallel Programming in C with MPI and OpenMP*