OpenMP

an Overview

Timothy H. Kaiser, Ph.D.
tkaiser@mines.edu

OpenMP talk

- What is it?
- Why are people interested?
- Why not?
- What does it look like?
- Examples please?
- Where to for more information

OpenMP

- OpenMP: An API for Writing Multithreaded Applications
  - Can be used create multi-threaded (MT) programs in Fortran, C and C++
  - Standardizes last 15-20 years of SMP practice

OpenMP

- Officially:
  - OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs.
- OpenMP Architecture Review Board: www.openmp.org, started in 1997

OpenMP

- OpenMP API uses the fork-join model of parallel execution
  - Works on a thread level
  - Works only on SMP machines
  - Directives placed in the source tell when to cause a forking of threads
    - Specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel
  - OpenMP-compliant implementations are not required to check for dependencies, conflicts, deadlocks, race conditions

OpenMP

- Directives:
  - Specify the actions to be taken by the compiler and runtime system in order to execute the program in parallel
  - OpenMP-compliant implementations are not required to check for dependencies, conflicts, deadlocks, race conditions

Why the Interest?

- Can be easy to parallelize an application
- We are starting to see commodity multi core machines

- Compilers are getting better
- Gcc and Gfortran support is ~~coming~~ here
- More efficient in memory usage?

Why not?

- SMP only - limits scaling
- Compilers are not that mature
- Easy to introduce bugs
- Thought of only for loop level parallelism (not true)
- Was first available for Fortran

How I got Involved

- Evaluation of IBM pre OpenMP compiler
- Hosted one of the OpenMP forum meetings
- Beat key compilers to death
  - Reported to vendors
  - Standards body
- Wrote OpenMP guide

Loop Directives

OpenMP and Directives

- OpenMP is a parallel programming system based on directives
- Directives are special comments that are inserted into the source to control parallel execution on a shared memory machine
- In Fortran all directives begin with !#OMP, C $OMP, or *$OMP
- For C they are #pragmas

For Fortran we have:

```
!#OMP parallel
C#OMP do parallel
*#OMP end parallel
```

For C we have:

```
#pragma parallel
#pragma for parallel
#pragma end parallel
```

A simple Example - Parallel Loop

```
!$OMP parallel do
  do n=1,128
    b(i) = a(i) + c(i)
  end do
!$OMP end parallel
```

- The first directive specifies that the loop immediately following should be executed in parallel. The second directive specifies the end of the parallel section
- For codes that spend the majority of their time executing loops the PARALLEL Do directive can result in significant parallel performance

Distribution of work SCHEDULE clause

The division of work among processors can be controlled with the SCHEDULE clause. For example

!$OMP parallel do schedule(STATIC)

Iterations are divided among the processors in contiguous chunks

!$OMP parallel do schedule(STATIC,N)

Iterations are divided round-robin fashion in chunks of size N

!$OMP parallel do schedule(DYNAMIC,N)

Iterations are handed out in chunks of size N as processors become available

Example

SCHEDULE(STATIC)

```
thread 0:do i=1,32                thread 2:do i=65,96
          a(i)=b(i)+c(i)                    a(i)=b(i)+c(i)
         enddo                             enddo

thread 1:do i=33,64               thread 3: do i=97,128
          a(i)=b(i)+c(i)                    a(i)=b(i)+c(i)
         enddo                             enddo
```

Note: With OpenMP version 3 static scheduling is deterministic

Example

SCHEDULE (STATIC,16)

```
thread 0: do i=1,16
           a(i)=b(i)+c(i)
          enddo
          do i=65,80
           a(i)=b(i)+c(i)
          enddo

thread 1: do i=17,32
           a(i)=b(i)+c(i)
          enddo
          do i=81,96
           a(i)=b(i)+c(i)
          enddo
```

```
thread 2: do i=33,48
           a(i)=b(i)+c(i)
          enddo
          do i=97,112
           a(i)=b(i)+c(i)
          enddo

thread3: do i=49,64
           a(i)=b(i)+c(i)
          enddo
          do i=113,128
           a(i)=b(i)+c(i)
          enddo
```

Private and Shared Data

SHARED - variable is shared by all processors
PRIVATE - each processor has a private copy of a variable

In the previous example of a simple parallel loop, we relied on the OpenMP defaults. Explicitly, the loop could be written as:

```
!$OMP parallel do SHARED(A,B,C,N) PRIVATE(I)
  do n=1,128
    b(i) = a(i) + c(i)
  end do
!$OMP end parallel
```

All processors have access to the same storage area for A, B, C, and N but each has its own private value for the loop index I.

Private data Example

In this loop each processor needs its own private copy of the variable TEMP. If TEMP were shared the result would be unpredictable

```
!$OMP parallel do SHARED(A,B,C,N) PRIVATE(I,TEMP)
  do i=1,N
    TEMP=A(i)/b(i)
    c(i) = TEMP + 1.0/TEMP
  end do
!$OMP end parallel
```

REDUCTION variables

Variables that are used in collective operations over the elements of an array can be labeled as REDUCTION variables.

```
ASUM = 0.0
APROD = 1.0
!$OMP PARALLEL DO REDUCTION (+:ASUM) REDUCTION (*:APROD)
do I=1,N
  ASUM = ASUM + A(I)
  APROD = APROD * A(I)
enddo
!$OMP END PARALLEL DO
```

Each processor has its own copy of ASUM and APROD. After the parallel work is finished, the master processor collects the values and performs a global reduction.

!$OMP Parallel alone

The !$OMP PARALLEL directive can be used to mark entire regions as parallel. The following two examples are equivalent.

```
!$OMP PARALLEL DO SCHEDULE (STATIC) firstprivate(a1,a2,a3,a4,a5)
    do j=j1,j2
        do i=i1,i2
            new_psi(i,j)=a1*psi(i+1,j)+a2*psi(i-1,j)+ &
                         a3*psi(i,j+1)+a4*psi(i,j-1)- &
                         a5*for(i,j)
        enddo
    enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO SCHEDULE (STATIC) private(i)
    do j=j1,j2
        do i=i1,i2
            psi(i,j)=new_psi(i,j)
        enddo
    enddo
!$OMP END PARALLEL DO
```

Or are they?

```
!$OMP PARALLEL
!$OMP DO SCHEDULE (STATIC) private(i)
firstprivate(a1,a2,a3,a4,a5)
    do j=j1,j2
        do i=i1,i2
            new_psi(i,j)=a1*psi(i+1,j)+a2*psi(i-1,j)+ &
                         a3*psi(i,j+1)+a4*psi(i,j-1)- &
                         a5*for(i,j)
        enddo
    enddo
!$OMP END  DO
!$OMP  DO SCHEDULE (STATIC) private(i)
    do j=j1,j2
        do i=i1,i2
            psi(i,j)=new_psi(i,j)
        enddo
    enddo
!$OMP END DO
!$OMP END PARALLEL
```

!$OMP Parallel

When a parallel region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.

By using the NOWAIT clause at the end of a loop the unnecessary synchronization of threads can be avoided

```
!$OMP PARALLEL
!$OMP DO
do i=1,n
   a(i)=b(i)+c(i)
enddo
!$OMP END DO NO WAIT
!$OMP DO
do i=1,n
   x(i)=y(i)+z(i)
enddo
!$OMP END DO
!$OMP END PARALLEL
```

Some other Directives

- !$OMP critical
  - Only one thread can be in a region at a time
- !$OMP single
  - Only one thread executes a block of code
- !$OMP master
  - Only the master thread executes a block of code

Critical

```
!$OMP parallel
        myt=omp_get_thread_num()
        write(*,*)"thread= ",myt," of ",OMP_GET_NUM_THREADS()
!$OMP end parallel
```

Could get..

```
 thread=  2 of 4
 thread=  1 of 4
 thread=  0 of 4
 thread=  3 of 4
```

Could get..

```
 thread=  3  of  4
 thread=  3  of  4
 thread=  3  of  4
 thread=  3  of  4
```

```
!$OMP parallel
!$OMP critical
        myt=omp_get_thread_num()
        write(*,*)"critical thread= ",myt
!$OMP end critical
!$OMP end parallel
```

```
 critical thread=  0
 critical thread=  2
 critical thread=  3
 critical thread=  1
```

Any other ideas on fixing this?

Parallel Sections

- There can be an arbitrary number of code blocks or sections.
- The requirement is that the individual sections be independent.
- Since the sections are independent they can be run in parallel.

```
#pragma omp parallel sections
 {
#pragma omp section
  {
  }
#pragma omp section
  {
  }
#pragma omp section
  {
  }
...
...
}
```

Four Independent Matrix Inversions

```
#pragma omp parallel sections
 {
#pragma omp section
        {
        system_clock(&t1_start);
        over(m1,n);
        over(m1,n);
        system_clock(&t1_end);
        e1=mcheck(m1,n,1);
        t1_start=t1_start-t0_start;
        t1_end=t1_end-t0_start;
         }
#pragma omp section
        {
        system_clock(&t2_start);
        over(m2,n);
        over(m2,n);
        system_clock(&t2_end);
        e2=mcheck(m2,n,2);
        t2_start=t2_start-t0_start;
        t2_end=t2_end-t0_start;
         }
#pragma omp section
        {
          system_clock(&t3_start);
          over(m3,n);
          over(m3,n);
          system_clock(&t3_end);
          e3=mcheck(m3,n,3);
          t3_start=t3_start-t0_start;
          t3_end=t3_end-t0_start;
        }
#pragma omp section
        {
          system_clock(&t4_start);
          over(m4,n);
          over(m4,n);
          system_clock(&t4_end);
          e4=mcheck(m4,n,4);
          t4_start=t4_start-t0_start;
          t4_end=t4_end-t0_start;
        }
}
```

Four Independent Matrix Inversions

```
printf("section 1 start time= %10.5g   end time= %10.5g  error= %g\n",t1_start,t1_end,e1);
printf("section 2 start time= %10.5g   end time= %10.5g  error= %g\n",t2_start,t2_end,e2);
printf("section 3 start time= %10.5g   end time= %10.5g  error= %g\n",t3_start,t3_end,e3);
printf("section 4 start time= %10.5g   end time= %10.5g  error= %g\n",t4_start,t4_end,e4);
```

```
[geight]% setenv OMP_NUM_THREADS 2
[geight]% ./a.out
section 1 start time= 0.00039494   end time=     1.3827  error= 3.43807e-07
section 2 start time= 0.00038493   end time=     1.5283  error= 6.04424e-07
section 3 start time=     1.3862   end time=     2.8165  error= 3.67327e-06
section 4 start time=     1.5319   end time=     3.0124  error= 3.42406e-06
[geight]%
```

!$task directive new to OpenMP 3.0

When a thread encounters a task construct, a task is generated from the code for the associated structured block. The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task.

!$omp task *[clause[[,] clause] ...]*
structured-block
!$omp end task

where *clause* is one of the following:

Note: the "if" clause could be used to determine if another task has completed

if(*scalar-logical-expression***)**
untied
default(private | firstprivate | shared | none)
private(*list***)**
firstprivate(*list***)**
shared(*list***)**

Tasks can be asynchronous, you can start a task and it might not finish until you do a taskwait or exit the parallel region.

section and task comparison

```
!$omp parallel sections

!$omp section
      t1_start=ccm_time()
      call invert(m1,n)
      call invert(m1,n)
      t1_end=ccm_time()
      e1=mcheck(m1,n,1)
      t1_start=t1_start-t0_start
      t1_end=t1_end-t0_start

!$omp section
      t2_start=ccm_time()
      call invert(m2,n)
      call invert(m2,n)
      t2_end=ccm_time()
      e2=mcheck(m2,n,2)
      t2_start=t2_start-t0_start
      t2_end=t2_end-t0_start
...
...
!$omp end parallel sections
```

```
e1=1;e2=1;e3=1;e4=1
!$omp parallel
!$omp single
!$omp task
      t1_start=ccm_time()
      call invert(m1,n)
      call invert(m1,n)
!$omp end task
      t1_end=ccm_time()
!     e1=mcheck(m1,n,1)
      t1_start=t1_start-t0_start
      t1_end=t1_end-t0_start

!$omp task
      t2_start=ccm_time()
      call invert(m2,n)
      call invert(m2,n)
!$omp end task
      t2_end=ccm_time()
!     e2=mcheck(m2,n,2)
      t2_start=t2_start-t0_start
      t2_end=t2_end-t0_start

...
...
!$omp end single
!$omp end parallel
```

for Intel need to add -liompprof5 option

section and task comparison

```
[tkaiser@compute-9-29 openmp]$ setenv OMP_NUM_THREADS 4
[tkaiser@compute-9-29 openmp]$ ./invertf
section    1 start time= 0.0000     end time= 2.4560     error=.56647E-04
section    2 start time= .80000E-02 end time= 2.3710     error=.57039E-03
section    3 start time= .11000E-01 end time= 2.4030     error=.76449E-04
section    4 start time= 0.0000     end time= 2.3650     error=.30831E-01

[tkaiser@compute-9-29 openmp]$ ./task
section    1 start time= .50941E+08 end time= .40000E-02 error=1.0000
section    2 start time= .50941E+08 end time= .40000E-02 error=1.0000
section    3 start time= .50941E+08 end time= .40000E-02 error=1.0000
section    4 start time= .50941E+08 end time= .40000E-02 error=1.0000
taskwait     start time= .50941E+08 end time= 2.4220
final errors  .56647E-04 .57039E-03 .76449E-04 .30831E-01

[tkaiser@compute-9-29 openmp]$ setenv OMP_NUM_THREADS 2
[tkaiser@compute-9-29 openmp]$ ./invertf
section    1 start time= .10000E-02 end time= 2.2560     error=.56647E-04
section    2 start time= 2.2600     end time= 4.4520     error=.57039E-03
section    3 start time= .30000E-02 end time= 2.2490     error=.76449E-04
section    4 start time= 2.2540     end time= 4.4630     error=.30831E-01

[tkaiser@compute-9-29 openmp]$ ./task
section    1 start time= .50941E+08 end time= 0.0000     error=1.0000
section    2 start time= .50941E+08 end time= 0.0000     error=1.0000
section    3 start time= .50941E+08 end time= 0.0000     error=1.0000
section    4 start time= .50941E+08 end time= 0.0000     error=1.0000
taskwait     start time= .50941E+08 end time= 4.4400
final errors  .56647E-04 .57039E-03 .76449E-04 .30831E-01
```

Thread Private

- Thread Private: Each thread gets a copy
- Useful for globals such as Fortran Common and Module variables
- Our somewhat convoluted example is interesting
- Brakes compilers, even though it is in the standards document
- Shows saving values between parallel sections
- Uses derived types
- Parallel without loops, higher level parallelizm

Thread Private

```
      module a22_module8
        type thefit
          sequence
          real val
          integer index
        end type thefit
        real, pointer :: work(:)
        type(thefit) bonk
        save work,bonk
!$omp   threadprivate(work,bonk)
      end module a22_module8


      subroutine sub1(n)
      use a22_module8
!$omp   parallel private(the_sum)
        allocate(work(n))
        call sub2(the_sum)
        write(*,*)the_sum
!$omp   end parallel
      end subroutine sub1
```

```
subroutine sub2(the_sum)
  use a22_module8
  use omp_lib
  work(:) = 10
  bonk%index=omp_get_thread_num()
  the_sum=sum(work)
  work=work/(bonk%index+1)
  bonk%val=sum(work)
end subroutine sub2

subroutine sub3(n)
use a22_module8
!$omp   parallel
 write(*,*)"bonk=",bonk%index,work,bonk%val
!$omp   end parallel
end subroutine sub3

program a22_8_good
  n = 10
  call sub1(n)
  write(*,*)"serial section"
  call sub3(n)
end program a22_8_good
```

Thread Private

```
[mbpro:~/programming/keep/openmp] tkaiser% setenv OMP_NUM_THREADS 4
[mbpro:~/programming/keep/openmp] tkaiser% ./domodule
   100.0000
   100.0000
   100.0000
   100.0000
 serial section
 bonk=           0   10.00000       10.00000       10.00000       10.00000
   10.00000       10.00000       10.00000       10.00000       10.00000
   10.00000       100.0000
 bonk=           1   5.000000       5.000000       5.000000       5.000000
   5.000000       5.000000       5.000000       5.000000       5.000000
   5.000000       50.00000
 bonk=           2   3.333333       3.333333       3.333333       3.333333
   3.333333       3.333333       3.333333       3.333333       3.333333
   3.333333       33.33334
 bonk=           3   2.500000       2.500000       2.500000       2.500000
   2.500000       2.500000       2.500000       2.500000       2.500000
   2.500000       25.00000
[mbpro:~/programming/keep/openmp] tkaiser%
```

Fourier Transform

- Used as a test of compilers and scheduling
- Generally gives good results with little effort
- Some surprises:
  - Compile fft routine separately
  - Static 64 - Static 63
  - See user guide

```
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
            do i=1,size
                call four1(a(:,i),size,isign)
            enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
PRIVATE(i,j,k,tmp)
            do k=1,size
                i=k
                do j=i,size
                    tmp=a(i,j)
                    a(i,j)=a(j,i)
                    a(j,i)=tmp
                enddo
            enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
            do i=1,size
                call four1(a(:,i),size,isign)
            enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
            do j=1,size
                a(:,j)=factor*a(:,j)
            enddo
!$OMP END PARALLEL DO
```

OpenMP Runtimes

2d optics program kernel (20 * 1024x1024 ffts with convolution)
Run on 4 processors of Cray T90 with compiler version 3.1.0.0
Run with and without OpenMP directives

source	options	CPU	Wallclock
no_omp_fft.f	none	126.9	130.3
no_omp_fft.f	-O3	110.1	111.8
no_omp_fft.f	-task3	110.2	110.4
omp_fft.f	none	123.6	38.5
omp_fft.f	-O3	111.5	34.4

Mac: 2 x 2.66 Dual-Core Intel Xeon = 1.38 sec

Environmental Variables

- OMP_NUM_THREADS
  - Sets the number of threads to use for parallel region
- OMP_SCHEDULE
  - Sets default schedule type
    - Static
    - Dynamic
    - Guided

Some Library Routines

- omp_get_num_threads
  - Returns the number of threads in the team executing the parallel region
- omp_get_max_threads
  - Returns the value of the nthreads-var internal control variable
- omp_get_thread_num
  - Returns the thread number

References

- www.openmp.org
- Examples
  - http:// geco.mines.edu/workshop
- My OpenMP Guide
  - ~~http://coherentcognition.com/projects/port/articles/openmp/guide/~~
  - In the openmp examples directory: openmp.pdf

Compilers

- Intel
  - Fortran : ifort,
  - C/C++ :icc
  - Option to support OpenMP
    - -openmp

Compilers

- Portland Group
  - Fortran : pgf77, pgf90
  - C/C++ :pgcc
  - Option to support OpenMP
  - -mp
  - pgifortref.pdf has good examples

Run Script

- Can only use a single node for OpenMP programs
- You don't need to use mpiexec

```
#!/bin/tcsh
#PBS -l nodes=1:ppn=8
#PBS -l walltime=00:20:00
#PBS -N testIO
#PBS -o outx8.$PBS_JOBID.pbs
#PBS -e errx8.$PBS_JOBID.pbs
#PBS -r n
#PBS -V
#--------------------
cd $PBS_O_WORKDIR

setenv OMP_NUM_THREADS 4

my_program
```

Don't run OpenMP programs on the front end lest you be shot

A Script with variables

```
#!/bin/csh
#PBS -l nodes=1:ppn=8
#PBS -l walltime=00:10:00
#PBS -N testIO
#PBS -o out.pbs
#PBS -e err.pbs
#PBS -r n
#PBS -V
#-------------------
cd $PBS_O_WORKDIR

foreach NUM (1 2 4)
    setenv OMP_NUM_THREADS $NUM
    echo "OMP_NUM_THREADS=" $OMP_NUM_THREADS
    echo "intel"
    ./invertc
    echo " "
end
```

Examples