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
  - Read Chapter 6

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?
- Intel MIC and GPUs, BGQ

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:

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

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

Could get..

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

```
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]% export 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@n7 openmp]$ export OMP_NUM_THREADS=4
[tkaiser@n7 openmp]$ ./invertf
section    1 start time= .10000E-02 end time= 10.107     error=.56647E-04
section    2 start time= .10000E-01 end time= 10.107     error=.57039E-03
section    3 start time= .18000E-01 end time= 10.122     error=.76449E-04
section    4 start time= .19000E-01 end time= 10.126     error=.30831E-01
[tkaiser@n7 openmp]$ ./task
section    1 start time=  57321838.7749999985 end time= .20000E-02 error=1.0000
section    2 start time=  57321838.7849999964 end time= .20000E-02 error=1.0000
section    3 start time=  57321838.7939999998 end time= .20000E-02 error=1.0000
section    4 start time=  57321838.7740000039 end time= .20000E-02 error=1.0000
taskwait     start time=  57321838.7719999999 end time= 10.151
final errors  .56647E-04 .57039E-03 .76449E-04 .30831E-01
[tkaiser@n7 openmp]$ export OMP_NUM_THREADS=2
[tkaiser@n7 openmp]$ ./invertf
section    1 start time= .10000E-02 end time= 10.089     error=.56647E-04
section    2 start time= 10.094     end time= 20.170     error=.57039E-03
section    3 start time= .10000E-01 end time= 10.089     error=.76449E-04
section    4 start time= 10.094     end time= 20.178     error=.30831E-01
[tkaiser@n7 openmp]$ ./task
section    1 start time=  57322060.0419999957 end time= .20000E-02 error=1.0000
section    2 start time=  57322070.1330000013 end time= .20000E-02 error=1.0000
section    3 start time=  57322070.1200000048 end time= .20000E-02 error=1.0000
section    4 start time=  57322060.0370000005 end time= .20000E-02 error=1.0000
taskwait     start time=  57322060.0349999964 end time= 20.178
final errors  .56647E-04 .57039E-03 .76449E-04 .30831E-01
[tkaiser@n7 openmp]$
```

Section and Task

```
!$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
```

```
!$omp parallel
!$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
```

Why "odd" times for t1_start?

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 parallelism

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()
        work=work/(bonk%index+1)
        the_sum=sum(work)
        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% export 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%
```

"Simplified"

```
      module mymod
        real, pointer :: work(:)
        save work,val,index
!$omp   threadprivate(work,val,index)
      end module mymod

!############################
      subroutine sub1(n)
      use mymod
      use omp_lib
!$omp   parallel private(the_sum,i)
        allocate(work(n))
        call sub2(the_sum)
        i=omp_get_thread_num()
        write(*,*)"from sub1",i,the_sum
!$omp   end parallel
      end subroutine sub1
!############################

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

```
!############################
      subroutine sub3(n)
      use mymod
!$omp   parallel
       write(*,*)"index=",index, &
                 " val=",val, &
                 " work=",work
!$omp   end parallel
      end subroutine sub3
!############################

      program a22_8_good
        n = 4
        call sub1(n)
        write(*,*)"serial section"
        call sub3(n)
      end program a22_8_good
```

Output

```
[tkaiser@n7 openmp]$ ./notype
 from sub1           0   40.00000
 from sub1           1   40.00000
 from sub1           2   40.00000
 from sub1           3   40.00000
 serial section
 index=           0  val=   40.00000      work=   10.00000 10.00000 10.00000 10.00000
 index=           3  val=   10.00000      work=   2.500000 2.500000 2.500000 2.500000
 index=           2  val=   13.33333      work=   3.333333 3.333333 3.333333 3.333333
 index=           1  val=   20.00000      work=   5.000000 5.000000 5.000000 5.000000
[tkaiser@n7 openmp]$
```

More Threadprivate

Each thread also has access to another type of memory that must not be accessed by other threads, called *threadprivate memory*.

Summary

The **threadprivate** directive specifies that variables are replicated, with each thread having its own copy.

Syntax

C/C++

The syntax of the **threadprivate** directive is as follows:

```
#pragma omp threadprivate(list) new-line
```

where *list* is a comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.

C/C++

Fortran

The syntax of the **threadprivate** directive is as follows:

```
!$omp threadprivate(list)
```

where *list* is a comma-separated list of named variables and named common blocks. Common block names must appear between slashes.

Fortran

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

Atomic

The advantage of using the **atomic** construct in this example is that it allows updates of two different elements of *x* to occur in parallel. If a **critical** construct were used instead, then all updates to elements of *x* would be executed serially (though not in any guaranteed order).

Note that the **atomic** directive applies only to the statement immediately following it.
As a result, elements of *y* are not updated atomically in this example.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
main() {
     float *x,*y,*work1,*work2;
     int *index;
     int n,i;
     n=10;
     x=(float*)malloc(n*sizeof(float));
     y=(float*)malloc(n*sizeof(float));
     work1=(float*)malloc(n*sizeof(float));
     work2=(float*)malloc(n*sizeof(float));
     index=(int*)malloc(10*sizeof(float));
     for( i=0;i < n;i++) {
          index[i]=(n-i)-1;
          x[i]=0.0;
          y[i]=0.0;
          work1[i]=i;
          work2[i]=i*i;
     }
#pragma omp parallel for  shared(x,y,index,n)
     for( i=0;i< n;i++) {
#pragma omp atomic
          x[index[i]] += work1[i];
          y[i] += work2[i];
     }
          for( i=0;i < n;i++)
                    printf("%d %g %g\n",i,x[i],y[i]);
}
```

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
- omp_get_wtime
  - Returns time in seconds

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
- https://computing.llnl.gov/tutorials/openMP

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

Compilers (BGQ - mc2)

- mpixlf90_r -qsmp=omp
- bgxlf90_r -qsmp=omp
- mpixlc_r -qsmp=omp
- bgxlc_r -qsmp=omp

Run Script

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

```
#!/bin/bash
#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

export OMP_NUM_THREADS=4

my_program
```

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

A Script with variables

For csh

```
#!/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
```

For bash (default on Mio)

```
#!/bin/bash
#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

for NUM in 1 2 4
    do
    export OMP_NUM_THREADS=$NUM
    echo "OMP_NUM_THREADS=" $OMP_NUM_THREADS
    echo "intel"
    ./invertc
    echo " "
   done
```

Run Script - mc2 (old)

```
#@job_name     = hybrid
#@comment      = “32 ranks per node"
#@output       = $(job_name)_$(jobid)_$(stepid).out
#@error        = $(job_name)_$(jobid)_$(stepid).err
#@environment  = COPY_ALL
#@job_type     = bluegene
#@notification = never
#@bg_size      = 1
#@bg_connectivity = torus
#@wall_clock_limit = 00:10:00
#@queue
cd $LOADL_STEP_INITDIR
JOBID=`echo $LOADL_STEP_ID | sed -e "s/mc2.local.//"`
ls
pwd

echo "trying runjob"
export OMP_NUM_THREADS=4
printenv OMP_NUM_THREADS > env_$JOBID
runjob --np 16 --ranks-per-node 8 --exp-env OMP_NUM_THREADS --exe /gpfs/sb/home/mc2/test/docol.exe
echo "got to the bottom"
```

Run Script - mc2

```
#!/bin/bash -x
#SBATCH --job-name="hybrid"
#comment= "glorified hello world"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --ntasks=8
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=10:00:00
#SBATCH -o ascript_out.%j

## Go to the directoy from which our job was launched
cd $SLURM_SUBMIT_DIR
JOBID=`echo $SLURM_JOBID`
mkdir ascript_$JOBID
cd ascript_$JOBID

echo "trying runjob"
export OMP_NUM_THREADS=4
printenv OMP_NUM_THREADS > env_$JOBID
srun --label  $SLURM_SUBMIT_DIR/docol.exe > output
srun --label  $SLURM_SUBMIT_DIR/docol.exe > output
echo "got to the bottom"
```

Examples

GPUs and MIC/Phi OpenMP?

Building for GPU with Cuda

- C extension
  - Write one portion of your program in regular C
    - Runs on CPU
    - Calls subroutines running on GPU
  - GPU code
    - Similar to regular C
    - Must pass in data from CPU
    - Must pay very close attention to data usage

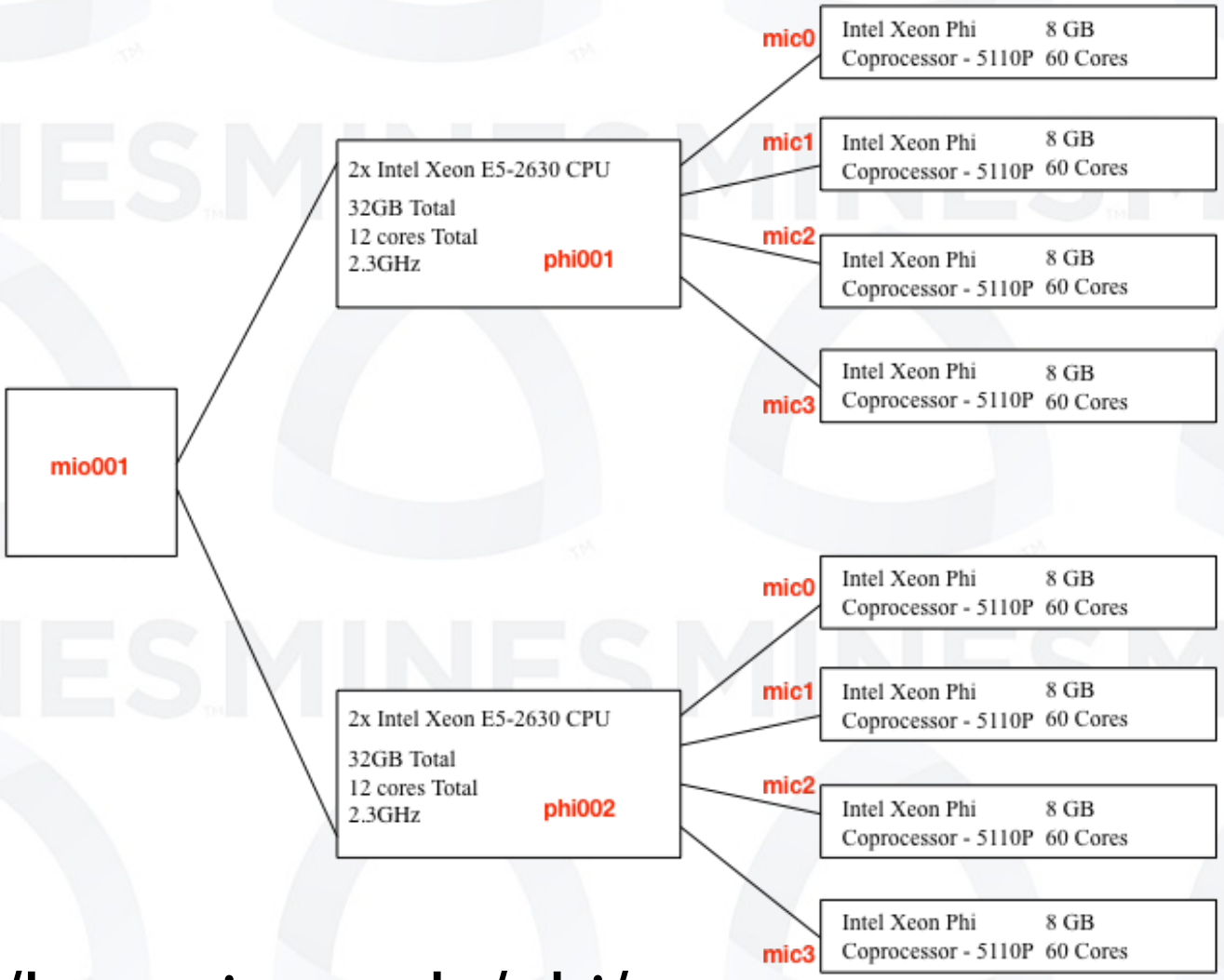
OpenACC

- Similar (more or less) to OpenMP
  - Directives can do:
    - Loops
    - Data distribution
  - http://www.openacc.org
  - Note: Portland Group was purchased by Nvidia

Intel MIC/Phi

- Top of the Top 500 list
- We have 2 nodes
  - 12 “normal” cores
  - 4 - MIC/Phi cards each with 60 cores

Intel MIC/Phi



http://hpc.mines.edu/phi/

Intel MIC/Phi

- They can run a (mini) full OS
- MIC has same instruction set as normal Intel chips
- Must still be compiled with different flags
- Binaries are not compatible
- However...

Intel MIC/ has several modes

MPI jobs

1. On card
2. Across multiple cards
3. With phi00x participating with one or more cards

Treading (OpenMP)

MKL

4. Programs that make calls to the MKL library running on the card
5. Offload - programs running on phi00x making MKL calls that are actually run on the card

Offload

6. Programs run on phi00x can call programs on the card
7. Programs run on phi00x call subroutines to run on the card.Can run MPI with some tasks on Phi and some on “normal” chip