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
• More efficient in memory usage?
• Intel Knights XXX and GPUs, BGQ
• High core count chips both Power and X86

http://www.openmp.org/resources/openmp-compilers/
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

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

```c
!$OMP parallel do schedule(STATIC)
  Iterations are divided among the processors in contiguous chunks
```

```c
!$OMP parallel do schedule(STATIC,N)
  Iterations are divided round-robin fashion in chunks of size N
```

```c
!$OMP parallel do schedule(DYNAMIC,N)
  Iterations are handed out in chunks of size N as processors become available
```
Example

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 1, 32</td>
<td>i = 65, 96</td>
</tr>
<tr>
<td>a(i) = b(i) + c(i)</td>
<td>a(i) = b(i) + c(i)</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 33, 64</td>
<td>i = 97, 128</td>
</tr>
<tr>
<td>a(i) = b(i) + c(i)</td>
<td>a(i) = b(i) + c(i)</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
</tbody>
</table>

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

thread 3: 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:

```plaintext
!$OMP parallel do shared(A,B,C,N) private(I)
    do i=1,n
        b(i) = a(i) + c(i)
    end do
!$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.

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

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

\[
\begin{align*}
\text{ASUM} &= 0.0 \\
\text{APROD} &= 1.0 \\
\text{\$OMP PARALLEL DO REDUCTION (+:ASUM) REDUCTION (*:APROD)} \\
\text{do I=1,N} \\
\text{\hspace{1em} ASUM = ASUM + A(I)} \\
\text{\hspace{1em} APROD = APROD * A(I)} \\
\text{\hspace{1em} enddo} \\
\text{\$OMP END PARALLEL DO}
\end{align*}
\]

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

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

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

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

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

Could get..

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

Any other ideas on fixing this?
Hello World

program hybrid
  implicit none
  integer OMP_GET_MAX_THREADS, OMP_GET_THREAD_NUM

!$OMP PARALLEL
!$OMP CRITICAL
  write(unit=*, fmt="(a,i2,a,i2i,)")" thread= ", OMP_GET_THREAD_NUM(), " &
    of ", OMP_GET_MAX_THREADS()

!$OMP END CRITICAL
!$OMP END PARALLEL
end program
program hybrid
   implicit none
   integer myid,ierr
   integer mylen,core
   integer, external :: findmycpu
   CHARACTER(len=255) :: myname
   integer OMP_GET_MAX_THREADS,OMP_GET_THREAD_NUM
   Call Get_environment_variable("SLURMD_NODENAME",myname)
   if(len_trim(myname) .eq. 0)then
      Call Get_environment_variable("HOSTNAME",myname)
   endif
   myid=0
   !$OMP PARALLEL
   !$OMP CRITICAL
   core=findmycpu()
   write(unit=*,fmt="(i4,a,a)",advance="no")myid," running on ",trim(myname)
   write(unit=*,fmt="(a,i2,a,i2i,a,i8)"wert=",OMP_GET_THREAD_NUM(),"," & 
   of ",OMP_GET_MAX_THREADS(), 
   " on core",core
   !$OMP END CRITICAL
   !$OMP END PARALLEL
end program

#include <utmpx.h>
int sched_getcpu();

int findmycpu_ ()
{
   int cpu;
   cpu = sched_getcpu();
   return cpu;
}
[tkaiser@mio001 openmp]$ export OMP_NUM_THREADS=8

[tkaiser@mio001 openmp]$ srun -n 1 --cpus-per-task=8 ./hello
srun: job 3996898 queued and waiting for resources
srun: job 3996898 has been allocated resources
  0 running on compute130 thread=  0 of  8 on core       7
  0 running on compute130 thread=  4 of  8 on core       0
  0 running on compute130 thread=  2 of  8 on core       2
  0 running on compute130 thread=  1 of  8 on core       3
  0 running on compute130 thread=  7 of  8 on core       1
  0 running on compute130 thread=  5 of  8 on core       6
  0 running on compute130 thread=  6 of  8 on core       4
  0 running on compute130 thread=  3 of  8 on core       5

[tkaiser@mio001 openmp]$
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.

```c
#pragma omp parallel sections
{
    #pragma omp section
    {
    }
    #pragma omp section
    {
    }
    #pragma omp section
    {
    }
    ...
    ...
```
Four Independent Matrix Inversions

```c
#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

```c
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);
```

```bash
[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.

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

where `clause` is one of the following:

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

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

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

```c
!$omp parallel sections
!$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
```

... ...

```c
!$omp end parallel sections
```

```c
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
```

... ...

```c
!$omp end single
!$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

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
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       100.0000
bonk=           1   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       33.33334
bonk=           3   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%
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:

```c
#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.

### Fortran

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

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

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

```fortran
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
do i=1,size
call four1(a(:,i),size,isign)
enddo
!$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

<table>
<thead>
<tr>
<th>source</th>
<th>options</th>
<th>CPU</th>
<th>Wallclock</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_omp_fft.f</td>
<td>none</td>
<td>126.9</td>
<td>130.3</td>
</tr>
<tr>
<td>no_omp_fft.f</td>
<td>-O3</td>
<td>110.1</td>
<td>111.8</td>
</tr>
<tr>
<td>no_omp_fft.f</td>
<td>-task3</td>
<td>110.2</td>
<td>110.4</td>
</tr>
<tr>
<td>omp_fft.f</td>
<td>none</td>
<td>123.6</td>
<td>38.5</td>
</tr>
<tr>
<td>omp_fft.f</td>
<td>-O3</td>
<td>111.5</td>
<td>34.4</td>
</tr>
</tbody>
</table>
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.

Note: Can’t get this to break on current Intel compilers
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 icpc
- OpenMP 4.0 C/C++/Fortran supported in version 15.0 and 16.0 compilers
- OpenMP 4.5 C/C++/Fortran supported in version 17.0 compilers
- Option to support OpenMP
  - -openmp
Compilers IBM Power

- Fortran: `xlf_r, xlf90_r`
- C/C++: `xlc_r, xlC_r`
- XL C/C++ for Linux V13.1.4 and XL Fortran for Linux V15.1.4 support OpenMP 3.1 and selected features in OpenMP 4.0.
- XL C/C++ for Linux V13.1.5 on little endian distributions and XL Fortran for Linux V15.1.15 on little endian distributions (available in Dec 2016) support OpenMP 3.1 and features in OpenMP 4.5 (include device constructs for offloading to NVIDIA GPU).
- Option to support OpenMP
  - `-qsmp=omp`
- `pgifortref.pdf` has good examples
Compilers Portland Group

- Fortran : pgf77, pgf90, pgfortran
- C/C++ : pgcc, pgc++
- Support for OpenMP 3.1 in all languages on x86 systems running Linux and OS X, OpenPOWER systems running Linux, and in Fortran and C on Windows/x86
- 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`
#!/bin/bash -x
#SBATCH --job-name="threaded"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=00:10:00
#SBATCH --constraint="core16|core20|core24|core28"
#SBATCH --cpus-per-task=16

module purge
module load PrgEnv/devtoolset-6
module load impi/gcc/latest

# Go to the directory from which our job was launched
cd $SLURM_SUBMIT_DIR

# Make a copy of our script
cat $0 > $SLURM_JOB_ID.src

#run using 16 cores
export OMP_NUM_THREADS=16

# run an application
srun --cpus-per-task=16 -n 1 $SLURM_SUBMIT_DIR/pointer.gcc < input
#!/bin/bash
#SBATCH --time=01:00:00
#SBATCH --partition=ppc
#SBATCH --overcommit
#SBATCH --exclusive
#SBATCH --nodelist=ppc002
#SBATCH --gres=gpu:4
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --export=ALL
#SBATCH --out=%J.out
#SBATCH --err=%J.msg

# Go to the directory from which our job was launched
cd $SLURM_SUBMIT_DIR

# Make a copy of our script
cat $0 > $SLURM_JOB_ID.src

# Set up our environment
source /etc/profile
module purge
module load XL
module load MPICH

# Run on 8 cores
export OMP_NUM_THREADS=8

srun ./a.out < input
Parallel Matrix Inversion

Doing inversion of “nrays” matrices

```fortran
write(17,*)"generating data for run",k," of ",kmax
call my_clock(stime)
tarf=1.0_b8
!$OMP PARALLEL DO PRIVATE(twod,i3,j)
do i=1,nrays
twod=>tarf(:,,:,i)
j=omp_get_thread_num()+1
do i3=1,msize
twod(i3,i3)=j+10.0_b8
enddo
call my_clock(etime)
write(17,'(" generating time=",f12.3," threads=",i3)')real(etime-stime,b8),maxthreads

write(17,*)"starting inverts"
call my_clock(stime)
!$OMP PARALLEL DO PRIVATE(twod)
do i=1,nrays
twod=>tarf(:,,:,i)
call my_clock(cnt1(i))
cALL DGESV( N, NRHS, twod, LDA, IPIVs(:,i), Bs(:,i), LDB, INFOs(i) )
call my_clock(cnt2(i))
write(17,'(i5,i5,3(f12.3))')i,infos(i),cnt2(i),cnt1(i),real(cnt2(i)-cnt1(i),b8)
enddo
call my_clock(etime)
ttime=real(etime-stime,b8)
if (ttime .lt. 0.0_b8) ttime=ttime+real(3600*24,b8)
write(17,'(" invert time=",f12.3")')ttime
```

Doing inversion of “nrays” matrices
Parallel Matrix Inversion

Doing inversion of “nrays” matrices

<table>
<thead>
<tr>
<th>thread</th>
<th>nrays</th>
<th>generating time</th>
<th>starting invert</th>
<th>invert time</th>
<th>invert time</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0</td>
<td>39960.686</td>
<td>39790.111</td>
<td>170.575</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>39960.860</td>
<td>39790.111</td>
<td>170.749</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>39961.308</td>
<td>39790.111</td>
<td>171.197</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>39961.313</td>
<td>39790.111</td>
<td>171.202</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>39961.721</td>
<td>39790.111</td>
<td>171.610</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>39962.628</td>
<td>39790.111</td>
<td>172.517</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>39964.000</td>
<td>39790.111</td>
<td>173.889</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>39972.701</td>
<td>39790.111</td>
<td>182.500</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>40129.076</td>
<td>39960.686</td>
<td>168.390</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>40129.256</td>
<td>39960.860</td>
<td>168.396</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>40130.318</td>
<td>39961.313</td>
<td>169.005</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>40130.628</td>
<td>39961.308</td>
<td>169.320</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>40130.691</td>
<td>39961.721</td>
<td>168.970</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>40133.456</td>
<td>39962.629</td>
<td>170.827</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>40136.002</td>
<td>39964.000</td>
<td>172.002</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>40152.241</td>
<td>39972.701</td>
<td>179.540</td>
<td></td>
</tr>
</tbody>
</table>

invert time = 711.282
```
/opt/utility/mapping

[tkaiser@mio001 aff]$ sbatch sbatch_mem
Submitted batch job 3996457

[tkaiser@mio001 aff]$ /opt/utility/mapping
ppc002
USER        PID %CPU S STIME %MEM COMMAND         PSR PSET    TID
tkaiser   70524  777 - 11:59  1.5 a.out             -    -      -
tkaiser   - 105 R 11:59     - -                 0    -  70524
tkaiser   - 96.0 R 11:59    - -                 1    -  70535
tkaiser   - 96.0 R 11:59    - -                 2    -  70536
tkaiser   - 96.0 R 11:59    - -                 3    -  70537
tkaiser   - 95.7 R 11:59    - -                 4    -  70538
tkaiser   - 95.7 R 11:59    - -                 5    -  70539
tkaiser   - 95.7 R 11:59    - -                 6    -  70540
tkaiser   - 95.7 R 11:59    - -                 7    -  70541

[tkaiser@mio001 aff]$ sbatch simple
Submitted batch job 3996459

[tkaiser@mio001 aff]$ /opt/utility/mapping
compute122
USER       PID %CPU S STIME %MEM COMMAND         PSR PSET   TID
tkaiser  32002  982 - 12:00  6.3 pointer.gcc       -    -     -
tkaiser      - 89.5 R 12:00    - -                15    - 32002
tkaiser      - 79.6 R 12:00    - -                 1    - 32018
tkaiser      - 79.6 R 12:00    - -                 9    - 32019
tkaiser      - 78.0 R 12:00    - -                10    - 32021
tkaiser      - 79.6 R 12:00    - -                 2    - 32020
tkaiser      - 79.6 R 12:00    - -                 3    - 32022
tkaiser      - 78.6 R 12:00    - -                 8    - 32023
tkaiser      - 77.6 R 12:00    - -                 4    - 32024
tkaiser      - 79.6 R 12:00    - -                12    - 32025
tkaiser      - 79.6 R 12:00    - -                 5    - 32026
tkaiser      - 79.6 R 12:00    - -                13    - 32027
tkaiser      - 79.6 R 12:00    - -                 6    - 32028
tkaiser      - 79.6 R 12:00    - -                14    - 32029
tkaiser      - 79.6 R 12:00    - -                 7    - 32030
tkaiser      - 78.6 R 12:00    - -                11    - 32031
tkaiser      - 77.6 R 12:00    - -                 0    - 32032
tkaiser      -  1.0 R 12:00    - -                 0    - 32038
```
If we take out `--cpus-per-task`

```bash
[tkaiser@mio001 aff]$ /opt/utility/mapping
ppc002

[tkaiser@mio001 aff]$ sbatch simple
Submitted batch job 3996462

[tkaiser@mio001 aff]$ sbatch sbatch_mem
Submitted batch job 3996463
```
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

mio001

2x Intel Xeon E5-2630 CPU
32GB Total
12 cores Total
2.3GHz

phi001

mic0
Intel Xeon Phi
8 GB
Coprocessor - 5110P
60 Cores

mic1
Intel Xeon Phi
8 GB
Coprocessor - 5110P
60 Cores

mic2
Intel Xeon Phi
8 GB
Coprocessor - 5110P
60 Cores

mic3
Intel Xeon Phi
8 GB
Coprocessor - 5110P
60 Cores

phi002

mic0
Intel Xeon Phi
8 GB
Coprocessor - 5110P
60 Cores

mic1
Intel Xeon Phi
8 GB
Coprocessor - 5110P
60 Cores

mic2
Intel Xeon Phi
8 GB
Coprocessor - 5110P
60 Cores

mic3
Intel Xeon Phi
8 GB
Coprocessor - 5110P
60 Cores

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