OpenMP

an Overview

Timothy H. Kaiser, Ph.D.
tkaiser2@nrel.gov

Slides: https://github.com/timkphd/slides

OpenMP talk

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

My repository...

```
git clone https://github.com/timkphd/examples

cd examples/openmp
   And
cd examples/hybrid
```

OpenMP

- OpenMP: An API for Writing Multithreaded Applications
  - Can be used create multi-threaded (MT) programs in Fortran, C and C++
  - Standardizes 20+ years of SMP practice
  - https://www.openmp.org

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 (mostly) on Shared memory 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 now see commodity multi core machines
- Compilers are getting better
- Gcc and Gfortran support
- More efficient in memory usage?
- High core count chips, ARM, Power and X86

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

Why not?

- Shared memory processor (smp) only - limits scaling
- Compilers are not that mature - different level of support
- 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 under contract to DoD

Disclaimer

- Most of these examples are from OpenMP version 2 with a few from version 3
- Current version in development is 5
- Most compilers support version 4+
- Still covers the basics

openmp.org

Reference Guides - OpenMP

Home | Specifications | Blog | Community | Resources | News & Events | About

Reference Guides

Home > Resources > Reference Guides

These 8- and 12-page documents provide a quick reference to OpenMP with section numbers that refer you to where you can find greater detail in the full specification.

OpenMP 5.0

- OpenMP 5.0 Reference Guide (May 2019) PDF (optimized for web view)
- OpenMP 5.0 Reference Guide (May 2019) PDF (optimized for local printing)
- OpenMP 5.0 Reference Guide (May 2019) Purchase Lulu.com print-on-demand hard copy
- OpenMP 5.0 Tasking Reference Guide (Sep 2019) PDF

OpenMP 4.5

- OpenMP 4.5 Reference Guide – C/C++ (Nov 2015) PDF
- OpenMP 4.5 Reference Guide – Fortran (Nov 2015) PDF

OpenMP 4.0

- OpenMP 4.0 Reference Guide – C/C++ (October 2013 PDF)
- OpenMP 4.0 Reference Guide – Fortran (October 2013 PDF)

https://www.openmp.org/resources/refguides/

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
```

Loop Directives

A simple Example - Parallel Loop

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

!$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
```

```
!$OMP PARALLEL
!$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
```

Or are they?

!$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?

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
```

Hello World #2

```
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)")" thread= ",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;
}
```

Output

```
[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]$
```

Digression

My example phostone.c is a hybrid MPI/OpenMP can be used to show mapping of tasks and thread to cores

```
el1:hybrid> ml intel-mpi/2018.0.3

el1:hybrid> mpiicc -qopenmp phostone.c -o phostone

el1:hybrid> export OMP_NUM_THREADS=4

el1:hybrid> srun --nodes=2 -n 4 --account=hpcapps --partition=debug --time=00:01:00 ./phostone -F -t 4
srun: job 3698523 queued and waiting for resources
srun: job 3698523 has been allocated resources
MPI VERSION Intel(R) MPI Library 2018 Update 3 for Linux* OS

task    thread             node name  first task    # on node  core
0000      0001               r1i7n35        0000         0000  0002
0000      0000               r1i7n35        0000         0000  0017
0000      0002               r1i7n35        0000         0000  0001
0000      0003               r1i7n35        0000         0000  0000
0001      0002               r1i7n35        0000         0001  0019
0001      0000               r1i7n35        0000         0001  0035
0001      0001               r1i7n35        0000         0001  0018
0001      0003               r1i7n35        0000         0001  0021
0002      0002               r4i7n35        0002         0000  0001
0002      0000               r4i7n35        0002         0000  0017
0002      0001               r4i7n35        0002         0000  0002
0002      0003               r4i7n35        0002         0000  0000
0003      0000               r4i7n35        0002         0001  0035
0003      0001               r4i7n35        0002         0001  0020
0003      0002               r4i7n35        0002         0001  0019
0003      0003               r4i7n35        0002         0001  0018
total time      4.000
el1:hybrid>
```

How could we make this easier to read placement of threads to cores?

Sort it!

```
el1:hybrid> cat out | sort -k6,6 -k3,3 | grep ^00
0000      0003              r103u21        0000         0000  0000
0002      0001              r103u23        0002         0000  0000
0000      0002              r103u21        0000         0000  0001
0002      0002              r103u23        0002         0000  0001
0000      0001              r103u21        0000         0000  0002
0002      0003              r103u23        0002         0000  0002
0000      0000              r103u21        0000         0000  0017
0002      0000              r103u23        0002         0000  0017
0001      0003              r103u21        0000         0001  0018
0003      0003              r103u23        0002         0001  0018
0001      0002              r103u21        0000         0001  0019
0003      0002              r103u23        0002         0001  0019
0001      0001              r103u21        0000         0001  0020
0003      0001              r103u23        0002         0001  0020
0001      0000              r103u21        0000         0001  0035
0003      0000              r103u23        0002         0001  0035
```

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
  - Broke early 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

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

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

ThreadPrivate

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

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

```
!##########################
      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
!##########################
```

- Work, val, index are threadprivate - Each thread gets/keeps its own copy
- Program calls sub1
- Sub1 calls sub2 which is run in parallel
- After sub1 we are in a serial section
- Sub3 contains another parallel section which prints our the values calculated

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]$
```

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(n*sizeof(float));
    for( i=0;i < n;i++) {
        index[i]=(rand() % n);
        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

Environmental Variables

- OMP_STACKSIZE
- OMP_STACKSIZE environment variable controls the size of the stack for threads created by the OpenMP implementation
- Needed to run VASP in MPI/OpenMP hybrid mode
  - OMP_STACKSIZE=40000
  - Default units are kilobytes

```
!$OMP PARALLEL PRIVATE(WQ) REDUCTION(+:SIF) REDUCTION(-:FORHF)
!$OMP DO SCHEDULE(STATIC) FIRSTPRIVATE(GWORK,CRHOLM,AUG_DES,POTFAK,W1,CDLM0,CDLM,CDIJ) &
!$OMP PRIVATE(MQ,LSHIFT,N,NGLB,WEIGHT,IDIR,ENL,ISPINOR,NI,NT,LMMAXC,NPRO,NIP,RTMP)
              mband: DO MQ=1,WHF%WDES%NBANDS
. . .
. . .
```

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

Compilers

- Fortran : ifort, gfortran
- C/C++ : icc icpc, gcc, g++
- OpenMP >4.5 supported in most common compilers
- Option to support OpenMP
  - -fopenmp (-qopenmp deprecated)
  - -mp for PGI/NVIDIA
- See https://www.openmp.org/resources/openmp-compilers-tools for a list

Simple Script

```
#!/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 --cpus-per-task=16

module purge
module load intel-mpi/2018.0.3

# Go to the directoy 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/phostone -F -t 10
```

Use phostone to see if you are getting the mapping you expect.

Environmental Variables to get good scaling

- https://github.com/NREL/HPC/tree/master/slurm
  - Slurm Examples Includes MPI, OpenMP, combined MPI/OpenMP
- https://nrel.github.io/HPC/2021/06/18/srun.html
  - *Using srun to launch applications under slurm*
  - More compact discussion about setting variables for OpenMP and MPI/OpenMP programs

GPUs and OpenMP?

Building for GPU with Cuda

- C extension
  - Write one portion of your program in regular C or Fortran
    - 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