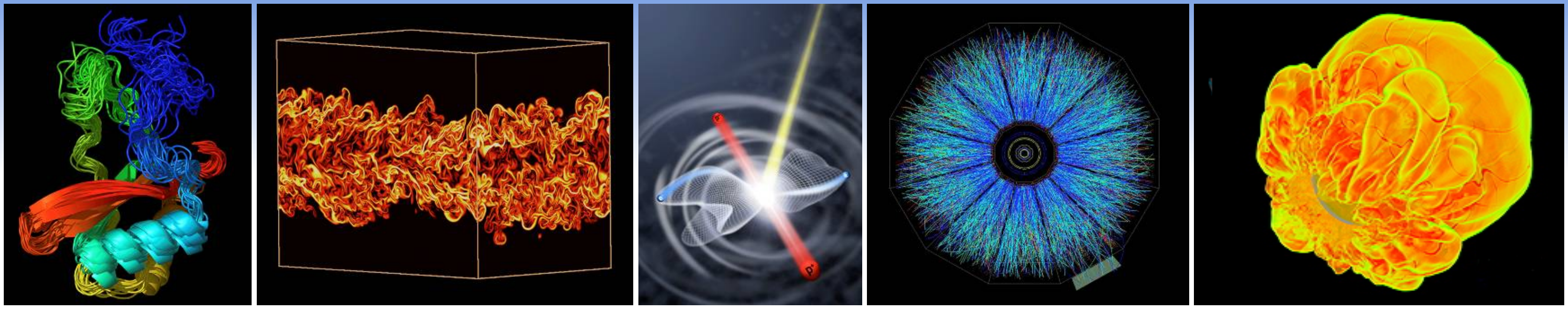

Introduction to OpenMP

Yun (Helen) He

NERSC User Group Meeting

Oct 18, 2010

Outline

- About OpenMP
- Parallel Regions
- Worksharing Constructs
- Synchronization
- Data Scope
- Tasks
- Using OpenMP at NERSC

Common Architectures

- Shared Memory Architecture
  - Multiple CPUs share global memory, could have local cache
  - Uniform Memory Access (UMA)
  - Typical Shared Memory Programming Model: OpenMP, Pthreads, …
- Distributed Memory Architecture
  - Each CPU has own memory
  - Non-Uniform Memory Access (NUMA)
  - Typical Message Passing Programming Model: MPI, …
- Hybrid Architecture
  - UMA within one SMP node
  - NUMA across nodes
  - Typical Hybrid Programming Model: mixed MPI/OpenMP, ...

What is OpenMP

- OpenMP is an industry standard API of C/C++ and Fortran for shared memory parallel programming.
  - OpenMP Architecture Review Board
    - Major compiler vendors: PGI, Cray, Intel, Oracle, HP, Fujitsu, Microsoft, AMD, IBM, NEC, Texas Instrument, …
    - Research institutions: cOMPunity, DOE/NASA Labs, Universities…
- History of OpenMP Standard
  - 1997 OpenMP 1.0 for Fortran, 1998 OpenMP 1.0 for C/C++
  - 2000 OpenMP 2.0 for Fortran, 2002 OpenMP 2.0 for C/C++
  - 2005 OpenMP 2.5 for all
  - 2008 OpenMP 3.0 for all
  - 2010 OpenMP 3.1 draft coming out soon

OpenMP Programming Model

- **Fork and Join Model**
  - **Master thread forks new threads at the beginning of parallel regions.**
  - **Multiple threads share work in parallel.**
  - **Threads join at the end of the parallel regions.**
- **Each thread works on global shared and its own private variables.**
- **Threads synchronize implicitly by reading and writing shared variables.**

Serial vs. OpenMP

Serial:
```
void main ()
{
   double x(256);
   for (int i=0; i<256; i++)
       {
       some_work(x[i]);
       }
}
```

OpenMP:
```
#include "omp.h"
Void main ()
{
   double x(256);
#pragma omp parallel for
   for (int i=0; i<256; i++)
       {
       some_work(x[i]);
       }
}
```

OpenMP is not just parallelizing loops!
It offers a lot more ….

Advantages of OpenMP

- Simple programming model
  - Data decomposition and communication handled by compiler directives
- Single source code for serial and parallel codes
- No major overwrite of the serial code
- Portable implementation
- Progressive parallelization
  - Start from most critical or time consuming part of the code

OpenMP Components

- Compiler Directives and Clauses
  - Interpreted when OpenMP compiler option is turned on.
  - Each directive applies to the succeeding structured block.
- Runtime Libraries
- Environment Variables

Compiler Directives

- Parallel Directive
  - Fortran: PARALLEL … END PARALLEL
  - C/C++: parallel
- Worksharing Constructs
  - Fortran: DO … END DO, WORKSHARE
  - C/C++: for
  - Both: sections
- Synchronization
  - master, single, ordered, flush, atomic
- Tasking
  - task, taskwait

Clauses

- private (list), shared (list)
- firstprivate (list), lastprivate (list)
- reduction (operator: list)
- schedule (method *[, chunk_size]*)
- nowait
- if (scalar_expression)
- num_thread (num)
- copyin (list)
- ordered
- collapse (n)
- tie, untie

OpenMP Runtime Libraries

- Number of threads
- Thread ID
- Scheduling
- Dynamic thread adjustment
- Nested Parallelism
- Active Levels
- Locking
- Wallclock timer

Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- OMP_ACTIVE_LEVELS
- OMP_THREAD_LIMIT

A Simple OpenMP Program

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main () {
    int tid, nthreads;
#pragma omp parallel private(tid)
   {
    tid = omp_get_thread_num();
    printf(”Hello World from thread %d\n", tid);
#pragma omp barrier
     if ( tid == 0 ) {
         nthreads = omp_get_num_threads();
         printf(”Total threads= %d\n",nthreads);
     }
  }
}
```

Sample Compile and Run:

```
% pgf90 –mp=nonuma test.f90
% setenv OMP_NUM_THREADS 4
% ./a.out
```

```
Program main
use omp_lib        (or: include “omp_lib.h”)
integer :: id, nthreads
!$OMP PARALLEL PRIVATE(id)
    id = omp_get_thread_num()
    write (*,*) ”Hello World from thread", id
!$OMP BARRIER
    if ( id == 0 ) then
        nthreads = omp_get_num_threads()
        write (*,*) "Total threads=",nthreads
     end if
!$OMP END PARALLEL
End program
```

Sample Output: (no specific order)

```
Hello World from thread          0
Hello World from thread          2
Hello World from thread          3
Hello World from thread          1
Total threads=        4
```

OpenMP Basic Syntax

- **Fortran: case insensitive**
  - **Add: use omp_lib or include “omp_lib.h”**
  - **Fixed format**
    - ***Sentinel* directive *[clauses]***
    - ***Sentinel* could be: !$OMP, *$OMP, c$OMP**
  - **Free format**
    - **!$OMP directive *[clauses]***
- **C/C++: case sensitive**
    - **Add: #include “omp.h”**
    - **#pragma omp directive *[clauses] newline***

The parallel Directive

FORTRAN:

```
!$OMP PARALLEL PRIVATE(id)
    id = omp_get_thread_num()
    write (*,*) "I am thread", id
!$OMP END PARALLEL
```

C/C++:

```
#pragma omp parallel private(thid)
{
     thid = omp_get_thread_num();
     printf("I am thread %d\n", thid);
}
```

- **The parallel directive forms a team of thread for parallel execution.**
- **Each thread executes within the OpenMP parallel region.**

Loop Parallelism

FORTRAN:

```
!$OMP PARALLEL [Clauses]
   …
!$OMP DO [Clauses]
    do i = 1, 1000
        a (i) = b(i) + c(i)
    enddo
!$OMP END DO [NOWAIT]
   …
!$OMP PARALLEL
```

C/C++:

```
#pragma omp parallel [clauses]
{   …
     #pragma omp for [clauses]
     {
        for (int i=0; i<1000; i++) {
            a[i] = b[i] + c[i];
        }
     }
…}
```

- **Threads share the work in loop parallelism.**
- **For example, using 4 threads under the default “static” scheduling, in Fortran:**
  - **thread 1 has i=1-250**
  - **thread 2 has i=251-500, etc.**

Combined Parallel Worksharing Constructs

FORTRAN:

```
!$OMP PARALLEL DO
    do i = 1, 1000
        a (i) = b(i) + c(i)
    enddo
!$OMP PARALLEL END DO
```

C/C++:

```
#pragma omp parallel for
     for (int i=0; i<1000; i++) {
         a[i] = b[i] + c[i];
     }
```

FORTRAN example:

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
     do i = 1, 1000
     c (i) = a(i) + b(i)
     enddo
!$OMP SECTION
     do i = 1, 1000
         d(i) = a(i) * b(i)
     enddo
!$OMP PARALLEL END SECTIONS
```

FORTRAN only:

```
INTEGER N, M
PARAMETER (N=100)
REAL A(N,N), B(N,N), C(N,N), D(N,N)
!$OMP PARALLEL WORKSHARE
      C = A + B
      M = 1
      D= A * B
!$OMP PARALLEL END WORKSHARE
```

Loop Parallelism: ordered and collapse

FORTRAN example:

```
!$OMP DO ORDERED
   do i = 1, 1000
      a (i) = b(i) + c(i)
   enddo
!$OMP END DO
```

FORTRAN example:

```
!$OMP DO COLLAPSE (2)
   do i = 1, 1000
      do j = 1, 100
         a(i,j) = b(i,j) + c(i,j)
      enddo
   enddo
!$OMP END DO
```

- **ordered specifies the parallel loop to be executed in the order of the loop iterations.**
- **collapse (*n*) collapse the *n* nested loops into 1, then schedule work for each thread accordingly.**

Loop-based vs. SPMD

Loop-based:

```
!$OMP PARALLEL DO PRIVATE(i)
!$OMP&            SHARED(a,b,n)
      do I =1, n
         a(i) = a(i) + b(i)
    enddo
!$OMP END PARALLEL DO
```

SPMD (Single Program Multiple Data):

```
!$OMP PARALLEL DO PRIVATE(start, end, i)
!$OMP&                  SHARED(a,b)
    num_thrds = omp_get_num_threads()
    thrd_id = omp_get_thread_num()
    start = n * thrd_id/num_thrds + 1
    end = n * (thrd_num+1)/num_thrds
    do i = start, end
        a(i) = a(i) + b(i)
    enddo
!$OMP END PARALLEL DO
```

SPMD code normally gives better performance than loop-based code, but is more difficult to implement:

- **Less thread synchronization.**
- **Less cache misses.**
- **More compiler optimizations.**

The barrier Directive

FORTRAN:
```
!$OMP PARALLEL
   do i = 1, n
       a(i) = b(i) + c(i)
   enddo
!$OMP BARRIER
   do i = 1, n
       e(i) = a(i) * d(i)
    enddo
!$OMP END PARALLEL
```

C/C++:
```
#pragma omp parallel
{   … some work;
     #pragma omp barrier
     … some other work;
}
```

- **Every thread waits until all threads arrive at the barrier.**
- **Barrier makes sure all the shared variables are (explicitly) synchronized.**

The critical Directive

FORTRAN:
```
!$OMP PARALLEL SHARED (x)
   … some work …
!$OMP CRITICAL [name]
    x = x + 1.0
!$OMP END CRITICAL
   … some other work …
!$OMP END PARALLEL
```

C/C++:
```
#pragma omp parallel shared (x)
{
#pragma omp critical
  {
  x = x +1.0;
  }
}
```

- **Each thread executes the critical region one at a time.**
- **Multiple critical regions with no name are considered as one critical region: single thread execution at a time.**

The master and single Directives

FORTRAN:

```
!$OMP MASTER
     ... some work ...
!$OMP END MASTER
```

C/C++:

```
#pragma omp master
{
   ... some work ...
}
```

FORTRAN:

```
!$OMP SINGLE
     ... some work ...
!$OMP END SINGLE
```

C/C++:

```
#pragma omp single
{
   ... some work ...
}
```

- **Master region:**
  - **Only the master threads executes the MASTER region.**
  - **No implicit barrier at the end of the MASTER region.**
- **Single region:**
  - **First thread arrives the SINGLE region executes this region.**
  - **All threads wait: implicit barrier at end of the SINGLE region.**

The atomic and flush Directives

FORTRAN:
```
!$OMP ATOMIC
    … some memory update …
```

C/C++:
```
#pragma omp atomic
  … some memory update …
```

FORTRAN:
```
!$OMP FLUSH [(var_list)]
```

C/C++:
```
#pragma omp flush [(var_list)]
```

- **Atomic:**
  - **Only applies to the immediate following statement.**
  - **Atomic memory update: avoids simultaneous updates from multiple threads to the same memory location.**
- **Flush:**
  - **Makes sure a thread's temporary view to be consistent with the memory.**
  - **Applies to all thread visible variables if no *var_list* is provided.**

Data Scope

- **Most variables are shared by default:**
  - **Fortran: common blocks, SAVE variables, module variables**
  - **C/C++: file scope variables, static**
  - **Both: dynamically allocated variables**
- **Some variables are private by default:**
  - **Certain loop indexes**
  - **Stack variables in subroutines or functions called from parallel regions**
  - **Automatic (local) variables within a statement block**

The firstprivate Clause

FORTRAN Example:
(from OpenMP spec 3.0)

```
PROGRAM MAIN
      INTEGER I, J
      I = 1
      J = 2
!$OMP  PARALLEL PRIVATE(I)
!$OMP& FIRSTPRIVATE(J)
      I = 3
      J = J + 2
!$OMP END PARALLEL
     PRINT*, I,J    ! I=1,J=2
END PROGRAM
```

- **Declares the variables in the list private**
- **Initializes the variables in the list with the value when they first enter the construct.**

The lastprivate Clause

FORTRAN example:
(from OpenMP spec 3.0)

```
program test
!$OMP parallel
!$OMP do private(j,k) collapse(2)
!$OMP& lastprivate(jlast, klast)
   do k = 1, 2
   do j = 1, 3
        jlast = j
        klast = k
   enddo
   enddo
!$OMP  end do
!$OMP single
   print *, klast, jlast     !prints 2 and 3
!$OMP end single
!$OMP end parallel
end program test
```

- **Declares the variables in the list private**
- **Updates the variables in the list with the value when they last exit the construct.**

The threadprivate and copyin Clauses

FORTRAN Example:
(from OpenMP spec 3.0)

```
      SUBROUTINE A25
        COMMON /T/ A
!$OMP THREADPRIVATE(/T/)

        CONTAINS
          SUBROUTINE B25
            COMMON /T/ A
!$OMP THREADPRIVATE(/T/)
            … some work …
!$OMP PARALLEL COPYIN(/T/)
!$OMP END PARALLEL
          END SUBROUTINE B25

      END SUBROUTINE A25
```

- **A threadprivate variable has its own copies of the global variables and common blocks.**
- **The copyin clause: copies the threadprivate variables from master thread to each local thread.**

The reduction Clause

C/C++ example:

```
    int i;
#pragma omp parallel reduction(*:i)
    {
      i=omp_get_num_threads();
     }
     printf(”result=%d\n”,i);
```

Fortran example:

```
        sum = 0.0
!$OMP parallel reduction (+: sum)
        do i =1, n
             sum = sum + x(i)
        enddo
!$OMP end do
!$OMP end parallel
```

- **Syntax: Reduction (operator : list).**
- **Reduces list of variables into one, using operator.**
- **Reduced variables must be shared variables.**
- **Allowed Operators:**
  - **Arithmetic: + - * / # add, subtract, multiply, divide**
  - **Fortran intrinsic: max min**
  - **Bitwise: & | ^ # and, or, xor**
  - **Logical: && || # and, or**

The schedule Clause

- Static: Loops are divided into *#thrds* partitions.
- Guided: Loops are divided into progressively smaller chunks until the chunk size is 1.
- Dynamic, *#chunk*: Loops are divided into chunks containing *#chunk* iterations.
- Auto: The compiler (or runtime system) decides what to use.
- Runtime: Use OMP_SCHEDULE environment variable to determine at run time.

The task and taskwait Directives

Serial:
```
int fib (int n)
{
   int x, y;
   if (n < 2) return n;
   x = fib (n – 1);
   y = fib (n – 2);
   return x+y;
}
```

OpenMP:
```
int fib (int n) {
   int x,y;
   if (n < 2) return n;
#pragma omp task shared (x)
   x = fib (n – 1);
#pragma omp task shared (y)
   y = fib (n – 2);
#pragma omp taskwait
   return x+y;
}
```

- **Major OpenMP 3.0 addition. Flexible and powerful.**
- **The task directive defines an explicit task.**
- **Threads share work from all tasks in the task pool.**
- **The taskwait directive makes sure all child tasks created for the current task finish.**

Some Runtime Functions

- omp_{set,get}_num_threads
- omp_get_thread_num
- omp_{set,get}_dynamic
- omp_in_parallel
- omp_{init,set,unset}_lock
- omp_get_thread_limit
- Timing routine: omp_get_wtime
  - thread private
  - call function twice, use difference between end time and start time

OMP_STACK_SIZE

- OMP_STACK_SIZE defines the private stack space each thread has.
- Default value is implementation dependent, and is usually quite small.
- Behavior is undefined if run out of space, mostly segmentation fault.
- To change, set OMP_STACK_SIZE to n (B,K,M,G) bytes. For example:

  setenv OMP_STACK_SIZE 16M

Compile OpenMP on Franklin and Hopper

- Use compiler wrappers:
  - ftn for Fortran codes
  - cc for C codes
  - CC for C++ codes
- Portland Group Compilers
  - Add compiler option “-mp=nonuma”
  - For example: % ftn –mp=nonuma mycode.f90
  - Supports OpenMP 3.0 from pgi/8.0

Compile OpenMP on Franklin and Hopper (2)

- Pathscale Compilers
  - % module swap PrgEnv-pgi PrgEnv-pathscale
  - Add compiler option “-mp”
  - For example: % ftn –mp=nonuma mycode.f90
- GNU Compilers
  - % module swap PrgEnv-pgi PrgEnv-gnu
  - Add compiler option “-fopenmp”
  - For example: % ftn –fopenmp mycode.f90
  - Supports OpenMP 3.0 from gcc/4.4

Compile OpenMP on Franklin and Hopper (3)

- Cray Compilers
  - % module swap PrgEnv-pgi PrgEnv-cray
  - No additional compiler option needed
  - For example: % ftn mycode.f90
  - Supports OpenMP 3.0

Run OpenMP on Franklin

- Each Franklin node has 4 cores with UMA.
- Use max 4 OpenMP threads per node.
- Interactive batch jobs:
  - Pure OpenMP example, using 4 OpenMP threads:
  - % qsub –I –V –q interactive
    –lmppwidth=1,mppnppn=1,mppdepth=4

    (Note: The above command should be in the same line)
  - wait for a new shell
  - % cd $PBS_O_WORKDIR
  - setenv OMP_NUM_THREADS 4
  - setenv PSC_OMP_AFFINITY FALSE *(note: for Pathscale only)*
  - % aprun –n 1 –N 1 –d 4 ./mycode.exe
- Change PBS mppwidth and aprun –n options to number of MPI tasks for hybrid MPI/OpenMP jobs.

Run OpenMP on Franklin (2)

Sample batch script:
(pure OpenMP example, using 4 OpenMP threads)

```
#PBS -q debug
#PBS -l mppwidth=1
#PBS -l mppnppn=1
#PBS -l mppdepth=4
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS –V
cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 4
aprun –n 1 -N 1 -d 4 ./mycode.exe
```

- **Run batch jobs:**
  - **Prepare a batch script first**
  - **Pure OpenMP example:**
  - **% qsub myscript**
- **If using pathscale:**
  - **setenv PSC_OMP_AFFINITY FALSE**
- **Hybrid MPI/OpenMP**
  - **2 Franklin nodes, 2 MPI tasks, 4 threads per MPI task:**
    - **request mppwidth=2**
    - **% aprun –n 2 –N 1 –d 4 ./mycode.exe**

Run OpenMP on Hopper

- This is about Hopper2, not the current Hopper1.
- Each Hopper node has 4 NUMA nodes, each with 6 UMA cores.
- Recommend to use max 6 OpenMP threads per NUMA node, and MPI across NUMA nodes. (although up to 24 OpenMP threads per Hopper node possible).
- Interactive batch jobs:
  - Pure OpenMP example, using 6 OpenMP threads:
  - % qsub –I –V –q interactive –lmppwidth=24
  - wait for a new shell
  - % cd $PBS_O_WORKDIR
  - setenv OMP_NUM_THREADS 6
  - setenv PSC_OMP_AFFINITY FALSE *(note: for Pathscale only)*
  - % aprun –n 1 –N 1 –d 6 ./mycode.exe
- Hybrid MPI/OpenMP:
  - 1 Hopper node, 4 MPI tasks, 6 OpenMP threads per MPI task:
  - % aprun –n 4 –N 4 –S 1 –ss –d 6 ./mycode.exe

-

Run OpenMP on Hopper (2)

Sample batch script:
(pure OpenMP example,
Using 6 OpenMP threads)

```
#PBS -q debug
#PBS -l mppwidth=24
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS –V
cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 6

#uncomment this line for pathscale
#setenv PSC_OMP_AFFINITY FALSE

aprun –n 1 -N 1 –d 6 ./mycode.exe
```

- **Run batch jobs:**
  - **Prepare a batch script first**
  - **Pure OpenMP example:**
  - **% qsub myscript**
- **Hybrid MPI/OpenMP**
  - **1 Hopper node, 4 MPI tasks, 6 OpenMP threads per MPI task:**
    - **% aprun –n 4 –N 4 –S 1 –ss –d 6 ./mycode.exe**
  - **2 Hopper nodes, 8 MPI tasks, 6 threads per MPI task:**
    - **#PBS -l mppwidth=48**
      - **24 cores/node *2 nodes**
    - **% aprun –n 4 –N 4 –S 1 –ss –d 6 ./mycode.exe**

Compile OpenMP on Carver

- Use compiler wrappers:
  - mpif90 for Fortran codes
  - mpicc for C codes
  - mpiCC for C++ codes
- Portland Group Compilers
  - Add compiler option “-mp=nonuma”
  - For example: % mpif90 –mp=nonuma mycode.f90
  - Supports OpenMP 3.0 from pgi/8.0

Compile on Carver (2)

- **GNU Compilers**
  - % module unload pgi openmpi
  - % module load gcc openmpi-gcc
  - Add compiler option "-fopenmp"
  - For example: % mpif90 –fopenmp mycode.f90
  - Supports OpenMP 3.0 from gcc/4.4
- **Intel Compilers**
  - % module unload pgi openmpi
  - % module load intel openmpi-intel
  - Add compiler option "-openmp"
  - For example: % mpif90 –openmp mycode.f90
  - Supports OpenMP 3.0 from intel/11.0

Run OpenMP on Carver

- Each Carver node has 8 cores with UMA.
- Use max 8 OpenMP threads per node.
- Interactive batch jobs:
  - Pure OpenMP example, using 8 OpenMP threads:
  - % qsub –I –V –q interactive –lnodes=1:ppn=1,pvmem=20GB
  - wait for a new shell
  - % cd $PBS_O_WORKDIR
  - setenv OMP_NUM_THREADS 8
  - % mpirun –np 1 ./mycode.exe
- Change PBS nodes:ppn, pvmem and mpirun –np options for hybrid MPI/OpenMP jobs.

Run OpenMP on Carver (2)

Sample batch script:
(pure OpenMP example,
using 4 OpenMP threads)

```
#PBS -q debug
#PBS -l nodes=1:ppn=1
#PBS –l pvmem=20GB
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS –V
cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 8
mpirun –np 1 ./mycode.exe
```

- **Run batch jobs:**
  - **Prepare a batch script first**
  - **Pure OpenMP example:**
  - **% qsub myscript**
- **Hybrid MPI/OpenMP**
  - **1 Carver node, 2 MPI tasks, 4 OpenMP threads per MPI task:**
    - **#PBS -l nodes=1:ppn=2**
    - **#PBS -l pvmem=10GB**
    - **Setenv OMP_NUM_THREADS 4**
    - **% mpirun –np 2 ./mycode.exe**
  - **2 Carver nodes, 2 MPI tasks, 8 threads per MPI task:**
    - **#PBS –l nodes=2:ppn=1**
    - **#PBS –l pvmem=20GB**
    - **Setenv OMP_NUM_THREADS 8**
    - **% aprun –np 2 ./mycode.exe**

Performance Results

Jacobi OpenMP	Execution Time (sec)	Speedup
1 thread	121	1
2 threads	63	1.92
4 threads	36	3.36

- Why not perfect speedup?
  - Serial code sections not parallelized
  - Thread creation and synchronization overhead
  - Memory bandwidth
  - Memory access with cache coherence
  - Load balancing
  - Not enough work for each thread

General Programming Tips

- Start from an optimized serial version.
- Gradually add OpenMP, check progress, add barriers.
- Decide which loop to parallelize. Better to parallelize outer loop. Decide whether loop permutation, fusion, exchange or collapse is needed.
- Use different OpenMP task scheduling options.
- Adjust environment variables.
- Choose between loop-based or SPMD.
- Minimize shared, maximize private, minimize barriers.
- Minimize parallel constructs, if possible use combined constructs.
- Take advantage of debugging tools: totalview, DDT, etc.

OpenMP *vs.* MPI

- Pure OpenMP Pro:
  - Easy to implement parallelism
  - Low latency, high bandwidth
  - Implicit Communication
  - Coarse and fine granularity
  - Dynamic load balancing
- Pure OpenMP Con:
  - Only on shared memory machines
  - Scale within one node
  - Possible data placement problem
  - No specific thread order

- Pure MPI Pro:
  - Portable to distributed and shared memory machines.
  - Scales beyond one node
  - No data placement problem
- Pure MPI Con:
  - Difficult to develop and debug
  - High latency, low bandwidth
  - Explicit communication
  - Large granularity
  - Difficult load balancing

Why Hybrid MPI/OpenMP

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures.
- Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes. Good usage of shared memory system resource (memory, latency, and bandwidth).
- Avoids the extra communication overhead with MPI within node.
- OpenMP adds fine granularity (larger message sizes) and allows increased and/or dynamic load balancing.
- Some problems have two-level parallelism naturally.
- Some problems could only use restricted number of MPI tasks.
- Possible better scalability than both pure MPI and pure OpenMP.

OpenMP Excersizes

- On NERSC machines: Franklin, Hopper2, and Carver:
  - % module load training
  - % cd $EXAMPLES/OpenMP/tutorial
- Try to understand, compile and run available examples.
  - Examples prepared by Richard Gerber, Mike Stewart, Helen He
- Have fun!

Further References

- OpenMP 3.0 specification, and Fortran, C/C++ Summary cards. http://openmp.org/wp/openmp-specifications/
- IWOMP2010 OpenMP Tutorial. Rudd van der Pas. http://www.compunity.org/training/tutorials/3%20Overview_OpenMP.pdf
- Shared Memory Programming with OpenMP. Barbara Chapman, at UCB 2010 Par Lab Boot Camp. http://parlab.eecs.berkeley.edu/sites/all/parlab/files/openmp-berkeley-chapman-slides_0.pdf
- SC08 OpenMP Tutorial. Tim Mattson and Larry Meadows. www.openmp.org/mp-documents/omp-hands-on-SC08.pdf
- Using OpenMP. Barbara Chapman, Gabrielle Jost, and Rudd van der Pas. Cambridge, MA: MIT Press, 2008.
- LLNL OpenMP Tutorial. Blaise Barney. http://computing.llnl.gov/tutorials/openMP
- NERSC OpenMP Tutorial. Richard Gerber and Mike Stewart. http://www.nersc.gov/nusers/help/tutorials/openmp
- Using Hybrid/OpenMP on NERSC Cray XT. Helen He. http://www.nersc.gov/nusers/systems/XT/openmp.php