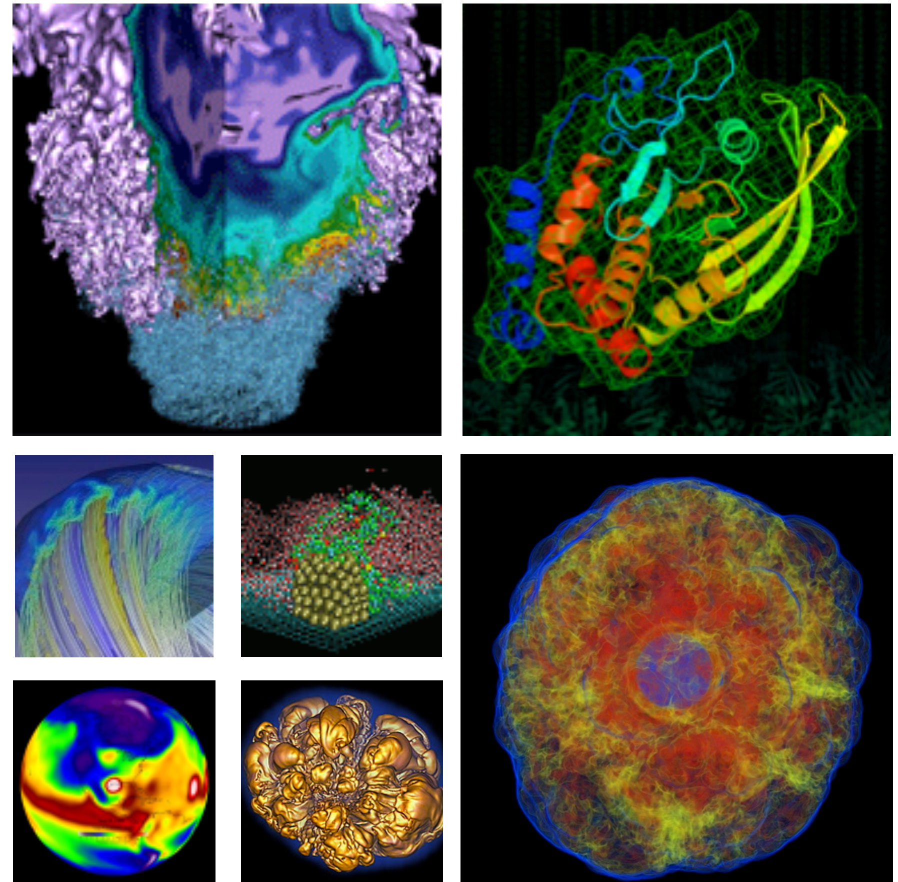
OpenMP Basics and MPI/OpenMP Scaling



Yun (Helen) He, NERSC

Mar 23, 2015

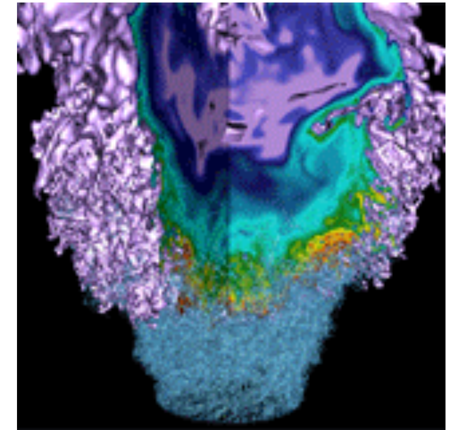
U.S. DEPARTMENT OF ENERGY | Office of Science

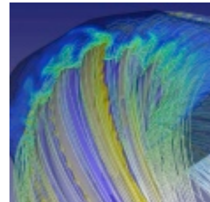
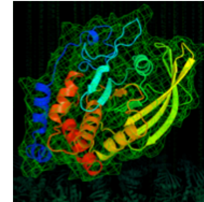
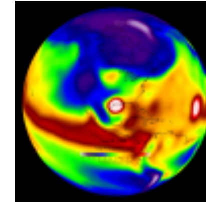
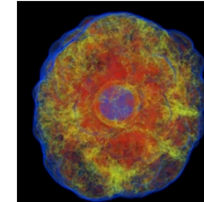
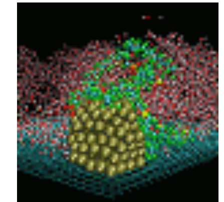
BERKELEY LAB

Outline

- Introductions to OpenMP
- Features in OpenMP 3.1
- What's new in OpenMP 4.0
- Adding OpenMP to your code using Cray Reveal
- Hybrid MPI/OpenMP Scaling

Introductions to OpenMP



U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

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 or socket
  - NUMA across nodes or sockets
  - Typical Hybrid Programming Model: hybrid MPI/OpenMP, ...

Current Architecture Trend

- **Multi-socket nodes with rapidly increasing core counts**
- **Memory per core decreases**
- **Memory bandwidth per core decreases**
- **Network bandwidth per core decreases**
- **Need a hybrid programming model with three levels of parallelism**
  - MPI between nodes or sockets
  - Shared memory (such as OpenMP) on the nodes/sockets
  - Increase vectorization for lower level loop structures

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: Intel, Cray, Intel, Oracle, HP, Fujitsu, NEC, NVIDIA, AMD, IBM, Texas Instrument, …
    - Research institutions: cOMPunity, DOE/NASA Labs, Universities…
  - The ARB mission is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.

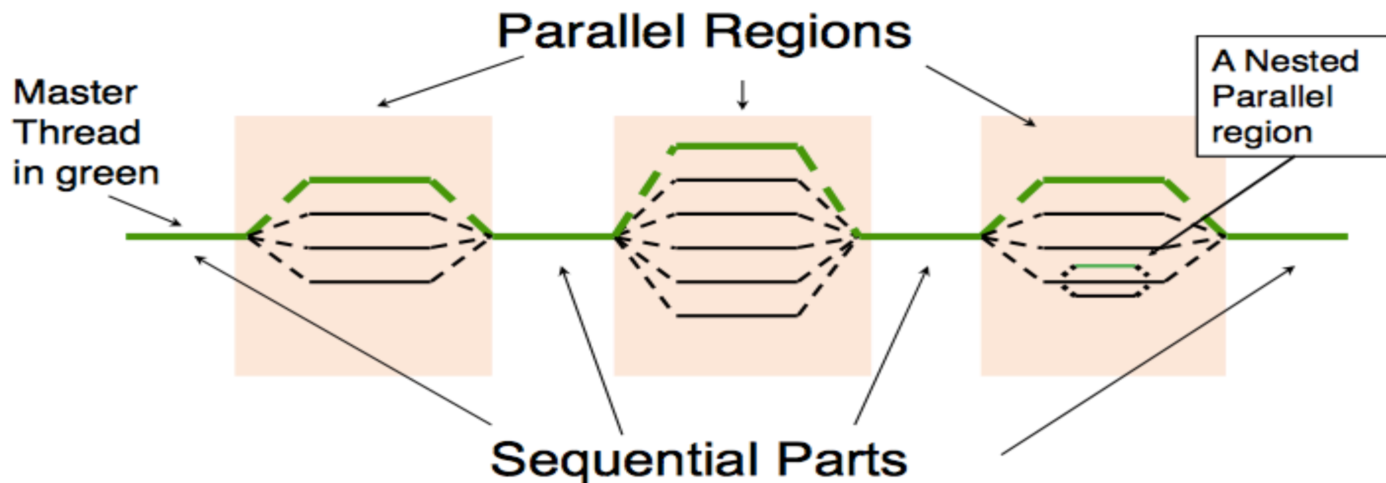
Please bring your OpenMP concerns to us

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**

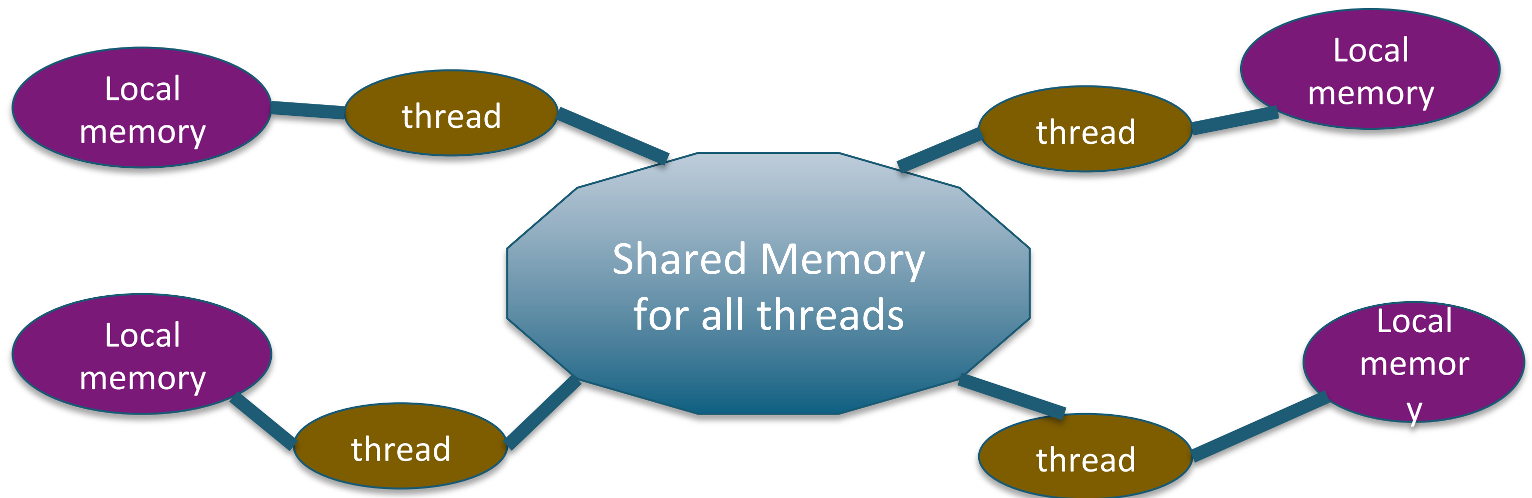
OpenMP Execution Model

- **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.



OpenMP Memory Model

- All threads have access to the same shared global memory.
- Each thread has access to its private local memory.
- Threads synchronize implicitly by reading and writing shared variables
- No explicit communication is needed between threads.



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 *vs.* MPI

- **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
- **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

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*

Compiler Directives

- **Parallel Direcitve**
  - Fortran: PARALLEL .... END PARALLEL
  - C/C++: paralllel
- **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_threads (num)
- threadprivate (list), copyin (list)
- ordered
- collapse (n)
- tie, untie
- And more ….

Runtime Libraries

- **Number of threads: omp_{set,get}_num_threads**
- **Thread ID: omp_get_thread_num**
- **Scheduling: omp_{set,get}_dynamic**
- **Nested parallelism: omp_in_parallel**
- **Locking: omp_{init,set,unset}_lock**
- **Active levels: omp_get_thread_limit**
- **Wallclock Timer: omp_get_wtime**
  - Thread private
  - Call function twice, use difference between end time and start time
- **And more …**

Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- OMP_ACTIVE_LEVELS
- OMP_THREAD_LIMIT
- And more ….

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:

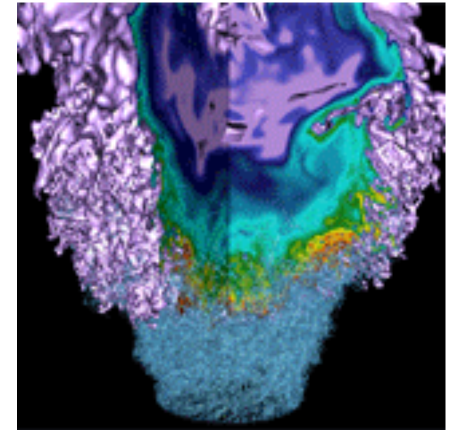
```
% ifort –openmp test.f90
% setenv OMP_NUM_THREADS 4
% ./a.out
```

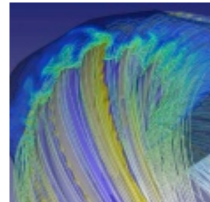
```
Program main
use omp_lib        (or: include “omp_lib.h”)
integer :: id, nthreads
!$OMP PARALLEL PRIVATE(id)
    id = omp_get_thread_num()
    write (*,*) ”Hello World from thread", id
!$OMP BARRIER
    if ( id == 0 ) then
        nthreads = omp_get_num_threads()
        write (*,*) "Total threads=",nthreads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

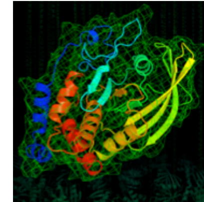
Sample Output: (no specific order)

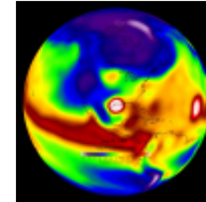
```
Hello World from thread        0
Hello World from thread        2
Hello World from thread        3
Hello World from thread        1
Total threads=        4
```

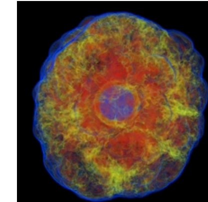
Features in OpenMP 3.1

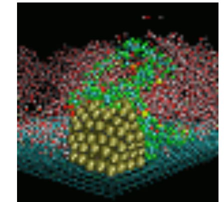












U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

Major Features in OpenMP 3.1

- Thread creation with shared and private memory
- Loop parallelism and work sharing constructs
- Dynamic work scheduling
- Explicit and implicit synchronizations
- Simple reductions
- Nested parallelism
- OpenMP tasking

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 threads for parallel execution.**
- **Each thread executes within the OpenMP parallel region.**

if and num_threads clauses

C/C++ example:

```
    int n=some_func();

#pragma omp parallel if(n>10)
    {
       … do_stuff;
     }
```

C/C++ example:

```
    int n=some_func();
#pragma omp parallel num_threads(n)
    {
       … do_stuff;
     }
```

- **The if clause contains a conditional expression. Fork only occurs if it is TRUE.**
- **The num_threads defines the number of threads active in a parallel construct**

First Hands-on Exercise

Get the Source Codes:

```
% cp –r /project/projecdirs/training/OpenMP_20150323/openmp .
```

Compile and Run:

```
% ftn –openmp –o hello_world hello_world.f90
(or % cc –openmp –o hello_world hello_world.c)
% qsub –I –q debug –lmppwidth=24
…
% cd $PBS_O_WORKDIR
% setenv OMP_NUM_THREADS 6 (for csh/tcsh)
  (or % export OMP_NUM_THREADS=6 for bash/ksh)
% aprun –n 1 –N 1 –d 6 ./hello_world
```

Sample Output: (no specific order)

```
Hello World from thread      0
Hello World from thread      2
Hello World from thread      4
Hello World from thread      3
Hello World from thread      5
Hello World from thread      1
Total threads=      6
```

Loop Parallelism

NERSC 40 YEARS at the FOREFRONT

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.**

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.

Second Hands-on Exercise

Sample codes: schedule.f90

-- Experiment with different number of threads.
-- Run this example multiple times.

```
% ftn –openmp –o schedule schedule.f90
% qsub –I –q debug –lmppwidth=24
…
% cd $PBS_O_WORKDIR
% setenv OMP_NUM_THREADS 3
% aprun –n 1 –N 1 –d 4 ./schedule
% setenv OMP_NUM_THREADS 6
…
```

-- Compare scheduling results with different scheduling algorithm: default static, "static, 2", "dynamic, 3", etc.
-- Results change with dynamic schedule at different runs.

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
```

Third Hands-on Exercise

Sample code: sections.f90

-- Experiment with different number of threads.
-- Run this example multiple times.

```
% ftn –openmp –o sections.f90
% qsub –V –I –q debug –lmppwidth=24
…
% cd $PBS_O_WORKDIR
% setenv OMP_NUM_THREADS 3
% aprun –n 1 –N 1 –d 3 ./sections
% setenv OMP_NUM_THREADS 5
% aprun –n 1 –N 1 –d 5 ./sections
```

-- What happens when more sections than threads?
-- What happens when more threads than sections?

Loop Parallelism: ordered and collapse Directives

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)* collapses the *n* nested loops into 1, then schedule work for each thread accordingly.**

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

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.**

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:**
  - **Arithmic: + - * / # add, substract, multiply, divide**
  - **Fortran intrinsic: max min**
  - **Bitwise: & | ^ # and, or, xor**
  - **Logical: && || # and, or**

Synchronization: barrier Directive

NERSC 40 YEARS at the FOREFRONT

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 will wait until all threads at the barrier.**
- **Barrier makes sure all the shared variables are (explicitly) synchronized.**

Synchronization: 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;
  }
}
```

- **Every 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.**

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Synchronization: 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.**

Synchronization: atomic and flush Directives

NERSC 40 YEARS at the FOREFRONT

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.

Thread Safety

- **In general, IO operations, general OS functionality, common library functions may not be thread safe. They should be performed by one thread only or serialized.**
- **Avoid race condition in OpenMP program.**
  - Race condition: Multiple threads are updating the same shared variable simultaneously.
  - Use “critical” directive
  - Use “atomic” directive
  - Use “reduction” directive

Fourth Hands-on Exercise

Sample codes: pi.c, pi_omp_wrong.c,
pi_omp1.c, pi_omp2.c, pi_omp3.c

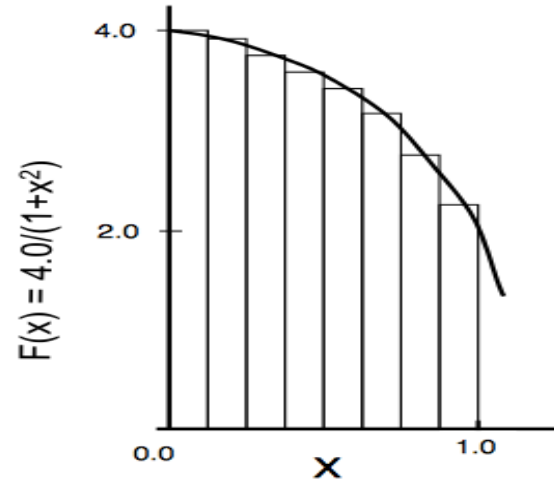
-- Understand different versions of calculating pi.
-- Understand the race condition in pi_omp_wrong.c
-- Run multiple times with different number of threads

```
% ./compile_pi
% qsub pi.pbs
```

-- Race condition generates different results.
-- Needs critical or atomic for memory updates.
-- Reduction is an efficient solution.

Serial code calculating PI

```
/* file name: pi.c */
/* this is the serial code of calculating pi */

static long num_steps = 100000;
double step;

int main()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0 /(double) num_steps;
   for (i=0;i<=num_steps; i++)
   {
     x=(i+0.5)*step;
     sum=sum+ 4.0/(1.0+x*x);
   }
   pi = step * sum;
   printf("pi=%f\n",pi);
   return 0;
}
```



Mathematically, we know:

$$\int_0^1 \frac{4.0}{(1+x^2)}dx = \pi$$

And this can be approximated as a sum of the area of rectangles:

$$\sum_{i=1}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has a width of Δx and a height of $F(x_i)$ at the middle of interval i.

pi_omp_wrong.c

```
/* pi_omp_wrong.c */

…
int main()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;

#pragma omp parallel private(x, sum)
{
#pragma omp for
   for (i=0;i<=num_steps; i++)
   {
     x=(i+0.5)*step;
     sum=sum+ 4.0/(1.0+x*x);
   }
   pi = pi + step * sum;
}
   printf("pi=%f\n",pi);
   return 0;
}
```

Multiple threads may update pi at the same time. Race condition!

pi_omp1.c with critical

```
/* file name: pi_omp1.c */

...
int main()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;

#pragma omp parallel private(x, sum)
{
#pragma omp for
   for (i=0;i<=num_steps; i++)
   {
     x=(i+0.5)*step;
     sum=sum+ 4.0/(1.0+x*x);
   }
#pragma omp critical
   pi = pi + step * sum;
}
   printf("pi=%f\n",pi);
   return 0;
}
```

"critical" directive ensures updates from one thread at a time

pi_omp2.c with atomic

```
/* file name: pi_omp2.c */

…
int main()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;

#pragma omp parallel private(x, sum)
{
#pragma omp for
   for (i=0;i<=num_steps; i++)
   {
     x=(i+0.5)*step;
     sum=sum+ 4.0/(1.0+x*x);
   }
#pragma omp atomic
   pi = pi + step * sum;
}
   printf("pi=%f\n",pi);
   return 0;
}
```

"atomic" does faster memory update than "critical". It also ensures update from one thread at a time

pi_omp3.c with reduction

```
/* file name: pi_omp3.c */

…
int main()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;

#pragma omp parallel for private(x) reduction (+:sum)
   for (i=0;i<=num_steps; i++)
   {
     x=(i+0.5)*step;
     sum=sum+ 4.0/(1.0+x*x);
   }
   pi = step * sum;
   printf("pi=%f\n",pi);
   return 0;
}
```

- "reduction" is a more scalable and elegant solution than "critical".
- Only 2 extra lines of codes than the serial code!

Data Scope

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

Data Sharing: firstprivate Clause

FORTRAN Example:

```
PROGRAM MAIN
      USE OMP_LIB
      INTEGER I
      I = 1
!$OMP  PARALLEL FIRSTPRIVATE(I) &
!$OMP PRIVATE(tid)
      I = I + 2    ! I=3
      tid = OMP_GET_THREAD_NUM()
      if (tid ==1) PRINT *, I    ! I=3
!$OMP END PARALLEL
     PRINT *, I    ! I=1
END PROGRAM
```

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

U.S. DEPARTMENT OF ENERGY | Office of Science
BERKELEY LAB

Data Sharing: lastprivate Clause

FORTRAN example:

```
Program main
Real A(100)
!$OMP parallel shared (A) &
!$OMP do lastprivate(i)
DO I = 1, 100
   A(I) = I + 1
ENDDO
!$OMP end do
!$OMP end parallel
    PRINT*, I       ! I=101
end program
```

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

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Data Sharing: threadprivate and copyin Clauses

FORTRAN Example:

```
PROGRAM main
 use OMP_LIB
      INTEGER tid, K
      COMMON /T/K
!$OMP THREADPRIVATE(/T/)
      K = 1

!$OMP  PARALLEL PRIVATE(tid) COPYIN(/T/)
      PRINT *, "thread ", tid, " ,K= ", K
      tid = omp_get_thread_num()
      K = tid + K
      PRINT *, "thread ", tid, " ,K= ", K
!$OMP END PARALLEL

!$OMP  PARALLEL PRIVATE(tid)
      tid = omp_get_thread_num()
      PRINT *, "thread ", tid, " ,K= ", K
!$OMP END PARALLEL
 END PROGRAM main
```

- A threadprivate variable has its own copies of the global variables and common blocks.
- A threadprivate variable has its scope across multiple parallel regions, unlike a private variable.
- The copyin clause: copies the threadprivate variables from master thread to each local thread.

Tasking: 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. Useful for situations other than counted loops.**
- **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.**
- **Helps to improve load balance.**

Nested OpenMP

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
   #pragma omp single
   {
      printf("Level %d: number of threads in the team - %d\n", level,
omp_get_num_threads());
   }
}
int main()
{
   omp_set_dynamic(0);
   #pragma omp parallel num_threads(2)
   {
      report_num_threads(1);
      #pragma omp parallel num_threads(2)
      {
         report_num_threads(2);
         #pragma omp parallel num_threads(2)
         {
            report_num_threads(3);
         }
      }
   }
   return(0);
}
```

```
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

```
% setenv OMP_NESTED TRUE
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
```

Level 0: P0
Level 1: P0 P1
Level 2: P0 P2; P1 P3
Level 3: P0 P4; P2 P5; P1 P6; P3 P5

U.S. DEPARTMENT OF ENERGY | Office of Science
BERKELEY LAB

Fifth Hands-on Exercise

Sample codes: jacobi_serial.f90 and jacobi_omp.f90

-- Check the OpenMP features used in the real code.
-- Understand code speedup.

```
% qsub jacobi.pbs
```

Or:

```
% ftn –openmp –o jacobi_omp.f90
% qsub –I –q debug –lmppwidth=24
…
% cd $PBS_O_WORKDIR
% setenv OMP_NUM_THREADS 6
% aprun –n 1 –N 1 –d 6 ./jacobi_omp
% setenv OMP_NUM_THREADS 12
% aprun –n 1 –N 1 –d 12 ./jacobi_omp
```

-- Why not perfect speedup?

Why not perfect speedup with OpenMP?

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

- **Possible causes**
  - 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

Cache Coherence and False Sharing

- ccNUMA node: cache-coherence NUMA node.
- Data from memory are accessed via cache lines.
- Multiple threads hold local copies of the same (global) data in their caches. Cache coherence ensures the local copy to be consistent with the global data.
- Main copy needs to be updated when a thread writes to local copy.
- Writes to same cache line from different threads is called false sharing or cache thrashing, since it needs to be done in serial. Use atomic or critical to avoid race condition.
- False sharing hurts parallel performance.

Cache Locality

- **Use data in cache as much as possible**
  - Use a memory stride of 1
    - Fortran: column-major order
    - C: row-major order
  - Access variable elements in the same order as they are stored in memory
  - Interchange loops or index orders if necessary
  - Tips often used in real codes

Fine Grain and Coarse Grain Models

NERSC 40 YEARS at the FOREFRONT

```
Program fine_grain
!$OMP PARALLEL DO
    do i=1,n
        … computation
    enddo
!$OMP END PARALLEL DO

  … some serial computation …

!$OMP PARALLEL DO
    do i=1,n
        … computation
    enddo
!$OMP END PARALLEL DO
end
```

- Program is single threaded except when actively using multiple threads, such as loop processing,
- Pro: Easier to adapt to MPI program.
- Con: thread overhead, serial section becomes bottleneck.

```
Program coarse_grain
!$OMP PARALLEL
!$OMP DO
    do i=1,n
        … computation
    enddo
!$OMP END DO

!$OMP DO
    do i=1,n
        … computation
    enddo
!$OMP END DO
!$OMP END PARALLEL
 end
```

- Majority of program run within an OMP parallel region.
- Pro: low overhead of thread creation, consistent thread affinity.
- Con: harder to code, prone to race condition.

Memory Affinity: "First Touch" Memory

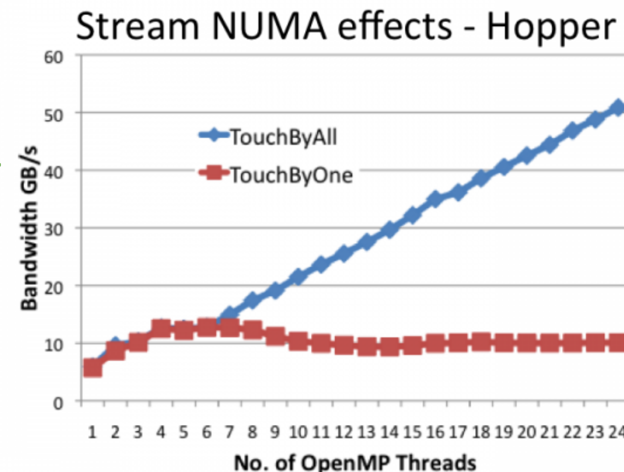
- **Memory affinity: allocate memory as close as possible to the core on which the task that requested the memory is running.**
- **Memory affinity is not decided by the memory allocation, but by the initialization. Memory will be local to the thread which initializes it. This is called "first touch" policy.**
- **Hard to do "perfect touch" for real applications. Instead, use number of threads few than number of cores per NUMA domain.**

Initialization

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

Compute

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j]=b[j]+d*c[j];}
```

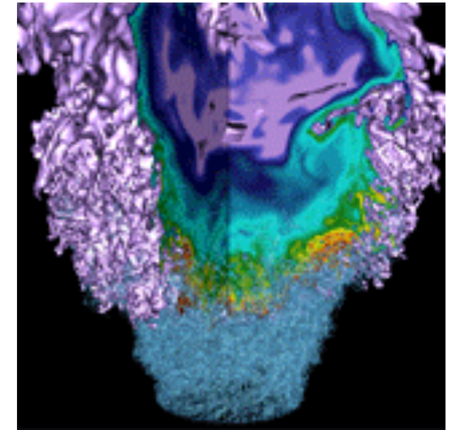


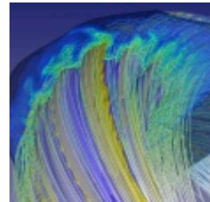
Courtesy of Hongzhang Shan

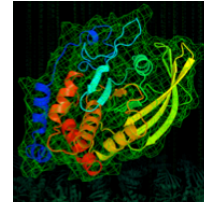
OMP_STACK_SIZE

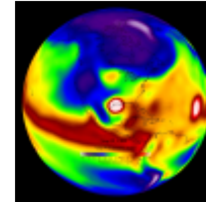
- OMP_STACK_SIZE defines the private stack space each thread has.
- Default value is implementation dependent, and is usually quite small.
- Behavior is undefined if run out of space, mostly segmentation fault.
- To change, set OMP_STACK_SIZE to n (B,K,M,G) bytes. setenv OMP_STACK_SIZE 16M

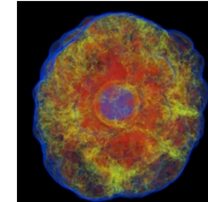
What's New in OpenMP 4.0

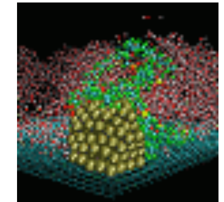












U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

New Features in OpenMP 4.0

- OpenMP 4.0 was released in July 2013
- Device constructs for accelerators
- SIMD constructs for vectorization
- Task groups and dependencies
- Thread affinity control
- User defined reductions
- Cancellation construct
- Initial support for Fortran 2003
- OMP_DISPLAY_ENV for all internal variables

Device Constructs for Accelerators

C/C++:
```
#pragma omp target map(to:B,C), map (tofrom: sum)
#pragma omp parallel for reduction(+,sum)
for (int i=0; i<N; i++) {
    sum += B[i] + C[i];
}
```

- Use "target" directive to offload a region to device.
- Host and device share memory via mapping: to, from, tofrom.

C/C++:
```
#pragma omp target teams distribute parallel for \
map (to:B,C), map (tofrom:sum) reduction(+:sum)
for (int i=0; i<N; i++) {
    sum += B[i] + C[i];
}
```

- Use "teams" clause to create multiple master threads that can execute in parallel on multiple processors on device.
- Use "distribute" clause to spread iterations of a parallel loop across teams.

OpenMP Vectorization Support

- More architectures support longer vector length.
- Vectorization: execute a single instruction on multiple data objects in parallel within a single CPU core.
- Auto-vectorization can be hard for compilers (dependencies).
- Many compilers support SIMD directives to aid vectorization of loops.
- OpenMP 4.0 provides a standardization for SIMD.

simd Directive

- **Vectorize a loop nest, cut loop into chunks that fit into a SIMD vector register**
- **Loop is not divided across threads**
- **Clauses can be:**
  - safelen(length): defines the max number of iterations can run concurrently without breaking dependence.
  - linear: lists variables with a linear relationship to the iteration number.
  - aligned: specifies byte alignment of the list items
  - all regular clauses ....

Parallelize and Vectorize a Loop Nest

NERSC 40 YEARS at the FOREFRONT

- C/C++: **#pragma omp for simd *[clauses]***
- Fortran: **!$OMP do simd *[clauses]***
- Divide loop first across a thread team, then subdivide loop chunks to fit a SIMD vector register.

SIMD Function Vectorization

C/C++:

```
#pragma omp declare simd
float min (float a, float b) {
    return a<b ? a:b;
}
```

- **Compilers may not be able to vectorize and inline function calls easily.**
- **#pramga declare simd tells compiler to generate SIMD function**
- **Useful to "declare simd" for elemental functions that are called from within a loop, so compilers can vectorize the function.**

taskgroup Directive

- OpenMP 4.0 extends the tasking support.
- The taskgroup directive waits for all descendant tasks to complete as compared to taskwait which only waits for direct children.

Task Dependencies

```
#pragma omp task depend (out:a)
{ ....}
#pragma omp task depend (out:b)
{...}
#pragma omp task depend (in:a,b)
{...}
```

- **The first two tasks can execute in parallel**
- **The third task can only start after both of the first two are complete.**

Better Thread Affinity Control

- **OpenMP 3.1 only has OMP_PROC_BIND, either TRUE or FALSE.**
- **OpenMP 4.0 still allows the above. Can now provide a list: master, close, or spread.**
- **Also added OMP_PLACES environment variable, can specify thread, cores, and sockets.**
- **Examples:**
  - export OMP_PLACES=threads
  - export OMP_PROC_BIND="spread, close" (for nested levels)

User Defined Reductions

```
#pragma omp declare reduction (merge: std::vector<int>
 : omp_out.insert(omp_out.end(), omp_in.begin(),
omp_in.end()))
```

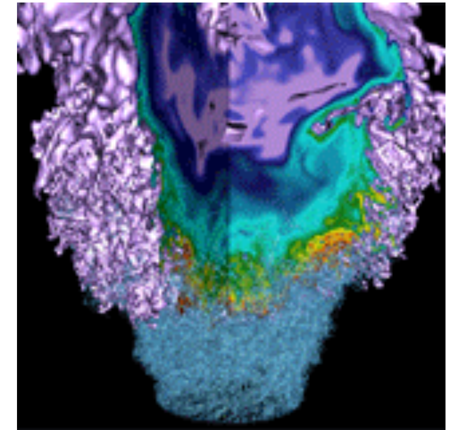
- **OpenMP 3.1 can not do reductions on objects or structures.**
- **OpenMP 4.0 can now define own reduction operations with declare reduction directive.**
- **"merge" is now a reduction operator.**

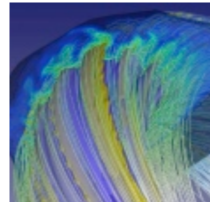
Construct Cancellation

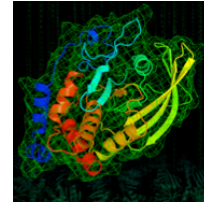
```
FORTRAN:
!$OMP PARALLEL DO PRIVATE (sample)
   do i = 1, n
      sample = testing(i,…)
!$OMP CANCEL PARALLEL IF (sample)
   enddo
!$OMP END PARALLEL DO
```

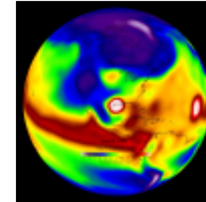
- **cancel / cancellation point is a clean way of early termination of an OpenMP construct.**
- **First thread exits when TRUE. Other threads exit when reaching the cancel directive.**

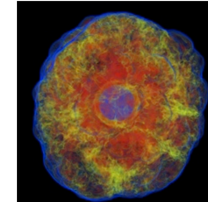
Adding OpenMP to Your Code Using Cray Reveal

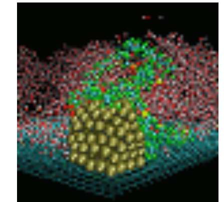












U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

What is Reveal

NERSC 40 YEARS at the FOREFRONT

- A tool developed by Cray to help developing the hybrid programming model (based on original serial or MPI code)
- Part of the Cray Perftools software package
- Only works under PrgEnv-cray
- Utilizes the Cray CCE program library for loopmark and source code analysis, combined with performance data collected from CrayPat
- Helps to identify top time consuming loops, with compiler feedback on dependency and vectorization
- Loop scope analysis provides variable scope and compiler directive suggestions for inserting OpenMP parallelism to a serial or pure MPI code

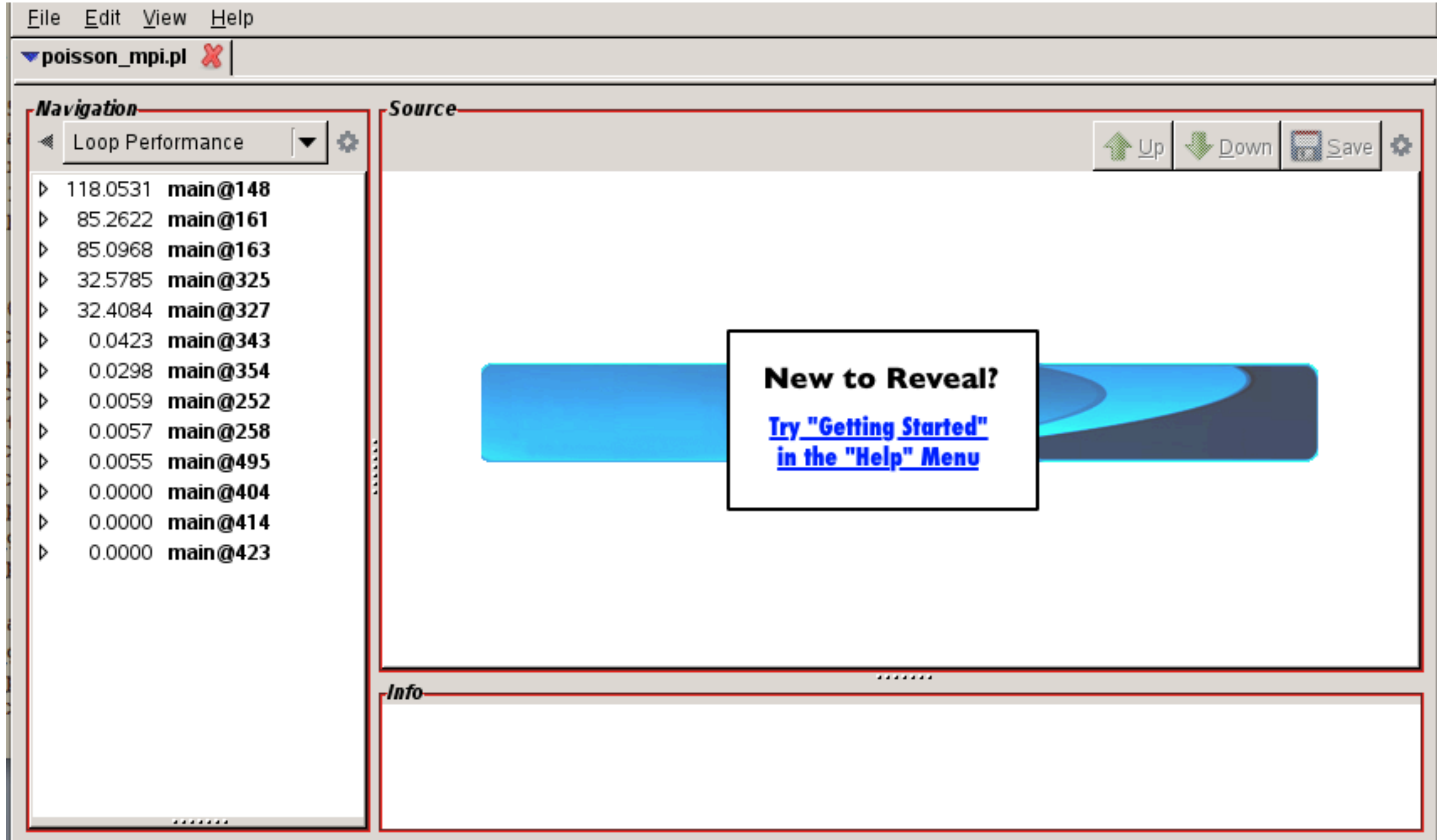
Steps to Use Reveal on Edison (1)

- **Load the user environment**
  - % module swap PrgEnv-intel PrgEnv-cray
  - % module unload darshan
  - % module load perftools (current default is version 6.2.2)
- **Generate loop work estimates**
  - % ftn –c –h profile_generate myprogram.f90
  - % ftn –o poisson_serial –h profile_generate myprogram.o
    - Good to separate compile and link to keep object files
    - Optimization flags disabled with –h profile-generate
  - % pat_build –w myprogram (-w enables tracing)
    - It will generate executable "myprogram+pat"
  - Run the program "myprogram+pat"
    - It will generate one or more myprogram+pat+…xf files
  - % pat_report myprogram+pat…xf > myprogram.rpt
    - It will generate myprogram+pat….ap2 file

Steps to Use Reveal on Edison (2)

- **Generate a program library**
  - % ftn –O3 –hpl=myprogram.pl –c myprogram.f90
  - Optimization flags can be used
  - Build one source code at a time, with "-c" flag
  - Use absolute path for program library if sources are in multiple directories
  - User needs to clean up program library from time to time
- **Launch Reveal**
  - % reveal myprogram.pl myprogram+pat…ap2

Cray Reveal GUI



Top loops with compiler loopmarks and feedback

Reveal

File Edit View Help

poisson_mpi.pl

Navigation

Loop Performance

118.0531	main@148
85.2622	main@161
85.0968	main@163
32.5785	main@325
32.4084	main@327
0.0423	main@343
0.0298	main@354
0.0059	main@252
0.0057	main@258
0.0055	main@495
0.0000	main@404
0.0000	main@414
0.0000	main@423

Top loops

Source dirs/yunhe/reveal/poisson_mpi.c

Up Down Save

Compiler loopmarks

```
    324  */
L   325    for ( i = i_min[my_rank] + 1; i <= i_max[my_rank] - 1; i++ )
    326    {
Lr4 327      for ( j = 1; j <= N; j++ )
    328      {
    329        u_new[INDEX(i,j)] =
    330          0.25 * ( u[INDEX(i-1,j)] + u[INDEX(i+1,j)] +
    331                   u[INDEX(i,j-1)] + u[INDEX(i,j+1)] +
    332                   h * h * f[INDEX(i,j)] );
    333      }
    334    }
    335  /*
    336    Wait for all non-blocking communications to complete.
    337  */
```

Info - Line 325

A loop was not vectorized because a recurrence was found between "u" and "u_new" at line 329.

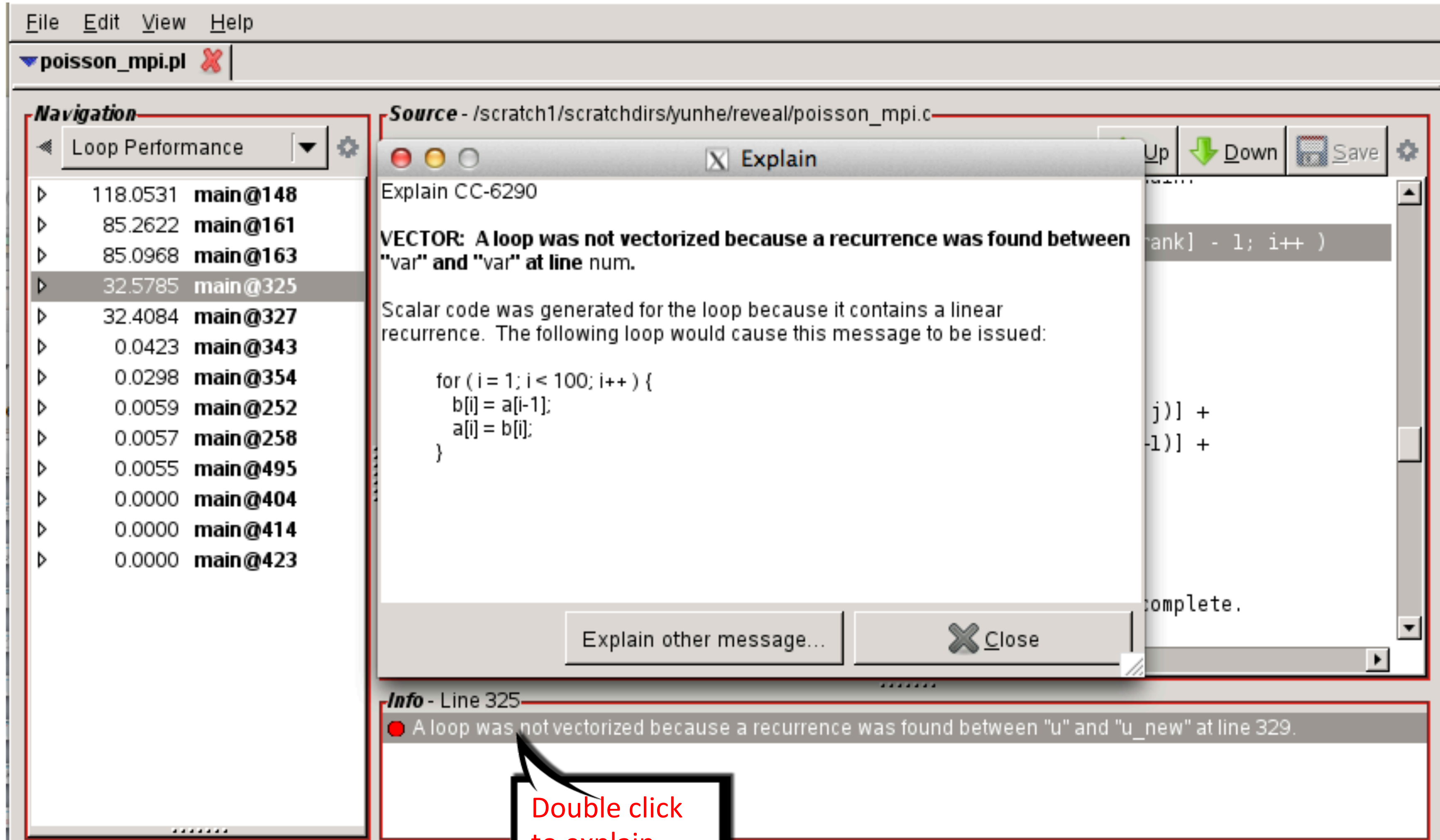
Compiler feedback

poisson_mpi.pl loaded. poisson_mpi+p... 2 loaded.

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Compiler feedback explanation



Compiler feedback explanation (2)

Reveal

File Edit View Help

poisson_mpi.pl

Navigation

Loop Performance

118.0531	main@148
85.2622	main@161
85.0968	main@163
32.5785	main@325
32.4084	main@327
0.0423	main@343
0.0298	main@354
0.0059	main@252
0.0057	main@258
0.0055	main@495
0.0000	main@404
0.0000	main@414
0.0000	main@423

Source - /scratch1/scratchdirs/yunhe/reveal/poisson_mpi.c

```
326   {
327     for ( j = 1; j <= N; j++ )
328     {
329       u_new[INDEX(i,j)] =
330         0.25 * ( u[INDEX(i-1,j)]
331                  u[INDEX(i,j-1)]
332                  h * h * f[INDEX
333     }
334   }
335 /*
336   Wait for all non-blocking comm
337 */
338   MPI_Waitall ( requests, reques
339 /*
```

Lr4

Info - Line 327

A loop was not vectorized because the loop initialization wou

A loop was unrolled 4 times.

poisson_mpi.pl loaded. poisson_mpi+pat+7119-5669t.ap2 loaded.

Explain

Explain CC-6005

SCALAR: A loop was unrolled.

The compiler unrolled the loop. Unrolling creates a number of copies of the loop body. When unrolling an outer loop, the compiler attempts to fuse replicated inner loops - a transformation known as unroll-and-jam. The compiler will always employ the unroll-and-jam mode when unrolling an outer loop; literal outer loop unrolling may occur when unrolling to satisfy a user directive (pragma).

This message indicates that unroll-and-jam was performed with respect to the identifed loop. A different message is issued when literal outer loop unrolling is performed, as this transfomation is far less likely to be beneficial.

For sake of illustration, the following contrasts unroll-and-jam with literal outer loop unrolling.

```
for ( j = 0; i < 10; i++ ) {
  for ( i = 0; i < 100; i++ ) {
    a[j][i] = b[j][i] + 42.0;
  }
}

for ( j = 0; i < 10; i +=2 ) {
  for ( i = 0; i < 100; i++ ) {
    a[j  ][i] = b[j  ][i] + 42.0;
    a[j+1][i] = b[j+1][i] + 42.0;
  }
}
```

Explain other message...

Close

Reveal scoping assistance

Reveal

File Edit View Help

poisson_mpi.pl

Navigation

Loop Performance

118.0531	main@148
85.2622	main@161
85.0968	main@163
32.5785	main@325
32.4084	main@327
0.0423	main@343
0.0298	main
0.0059	main@252
0.0057	main@258
0.0055	main@495
0.0000	main@404
0.0000	main@414
0.0000	main@423

Scope Loop

Source - /scratch1/scratchdirs/yunhe/reveal/poisson_mpi.c

Up Down Save

```
326  {
327    for ( j = 1; j <= N; j++ )
328    {
329      u_new[INDEX(i,j)] =
330        0.25 * ( u[INDEX(i-1,j)] + u[INDEX(i+1,j)]
331                 u[INDEX(i,j-1)] + u[INDEX(i,j+1)]
332                 h * h * f[INDEX(i,j)] );
333    }
334  }
          t for all non-blocking communications to compl
338  MPI_Waitall ( requests, request, status );
339  /*
```

Right click to select loops

Info - Line 343

A loop was not vectorized because the loop initialization would be too costly.

A loop was unrolled 4 times.

poisson_mpi.pl loaded. poisson_mpi+pat+7119-5669t.ap2 loaded.

Reveal OpenMP Scoping

Scope Loops | Scoping Results

Edit List | List of Loops to be Scoped

Scope?	Line #	File or Source Line
☑		/scratch1/scratchdirs/yunhe/reveal/poisson_mpi.c
☑	327	for (j = 1; j <= N; j++)
☑	343	for (j = 1; j <= N; j++)

Apply Filter Time: 0.000 Trips: 0

Start Scoping Cancel Close

Start Scoping

Scoping Results

Reveal

Help

ance

n@148
n@161
n@163
n@325
n@327
n@343
n@354
n@252
n@258
n@495
n@404
n@414
n@423

Source - /scratch1/scratchdirs/yunhe/reveal/poisson_mpi.c

Up Down Save

```
324 */
325   for ( i = i_min[my_rank] + 1; i <= i_max[my_rank] - 1; i++
326   {
327     for ( j = 1; j <= N; j++ )
328     {
329       u_new[INDEX(i,j)] =
330         0.25 * ( u[INDEX(i-1,j)] + u[INDEX(i+1,j)] +
331                  u[INDEX(i,j-1)] + u[INDEX(i,j+1)] +
332                  h * h * f[INDEX(i,j)] );
333     }
334   }
335 /*
336   Wait for all non-blocking communications to complete.
337 */
338   MPI_Waitall ( requests, request, status );
339 /*
```

Info - Line 325

A loop was not vectorized because a recurrence was found between "u" and "u_new" at line 329.

ded. poisson_mpi+pat+7119-5669t.ap2 loaded.

Reveal OpenMP Scoping

Scope Loops | Scoping Results

poisson_mpi.c: Loop@325

Name	Type	Scope	Info
f	Scalar	Unresolved	FAIL: Possible recurrence involving
			FAIL: Possible resolvable recurrenc
u	Scalar	Unresolved	FAIL: Possible recurrence involving
			FAIL: Possible resolvable recurrenc
u_new	Scalar	Unresolved	FAIL: Possible recurrence involving
			FAIL: Possible resolvable recurrenc
j	Scalar	Private	
N	Scalar	Shared	
h	Scalar	Shared	
i	Scalar	Shared	
i_max	Scalar	Shared	
i_min	Scalar	Shared	
my_rank	Scalar	Shared	

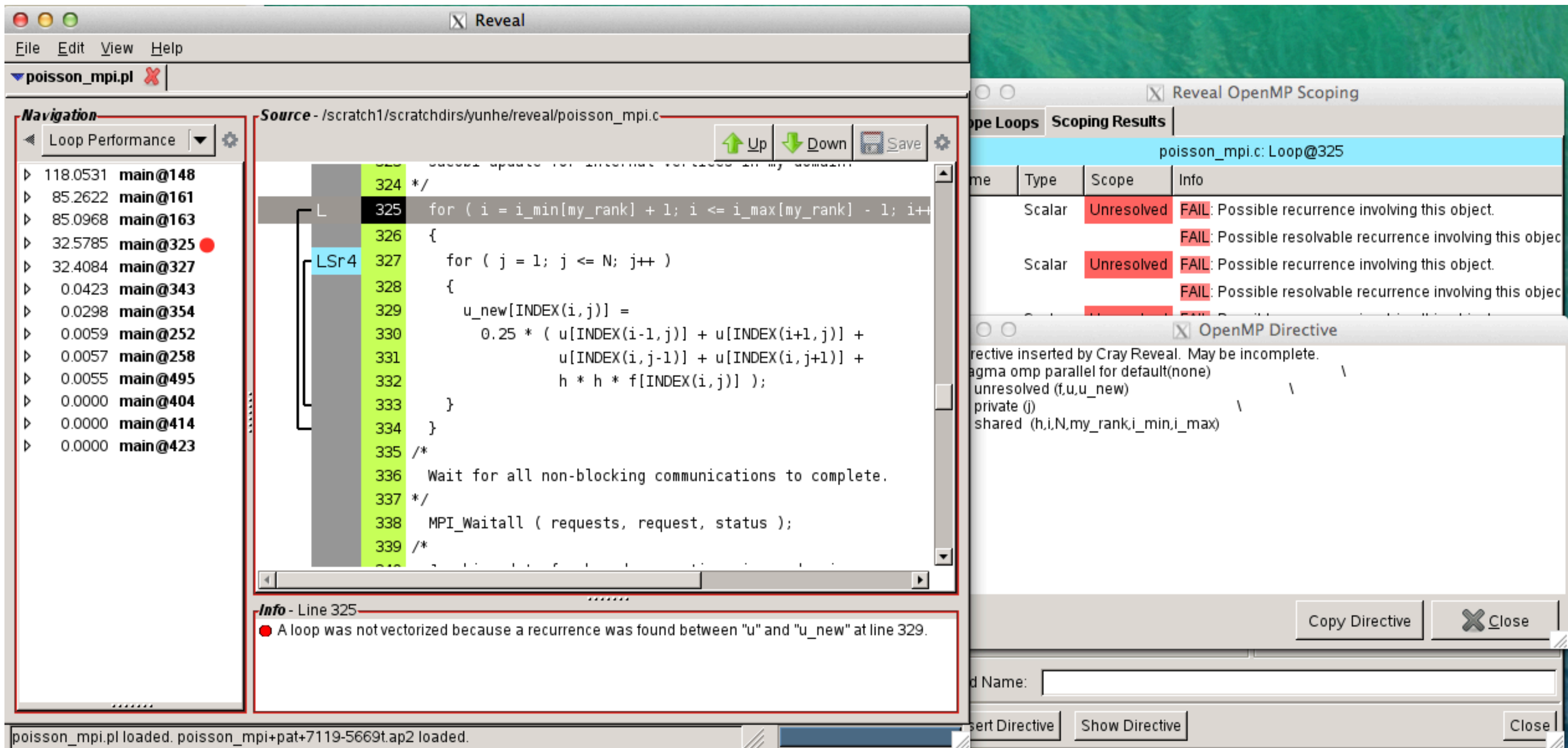
First/Last Private
- Enable FirstPrivate
- Enable LastPrivate

Reduction: None

Find Name:

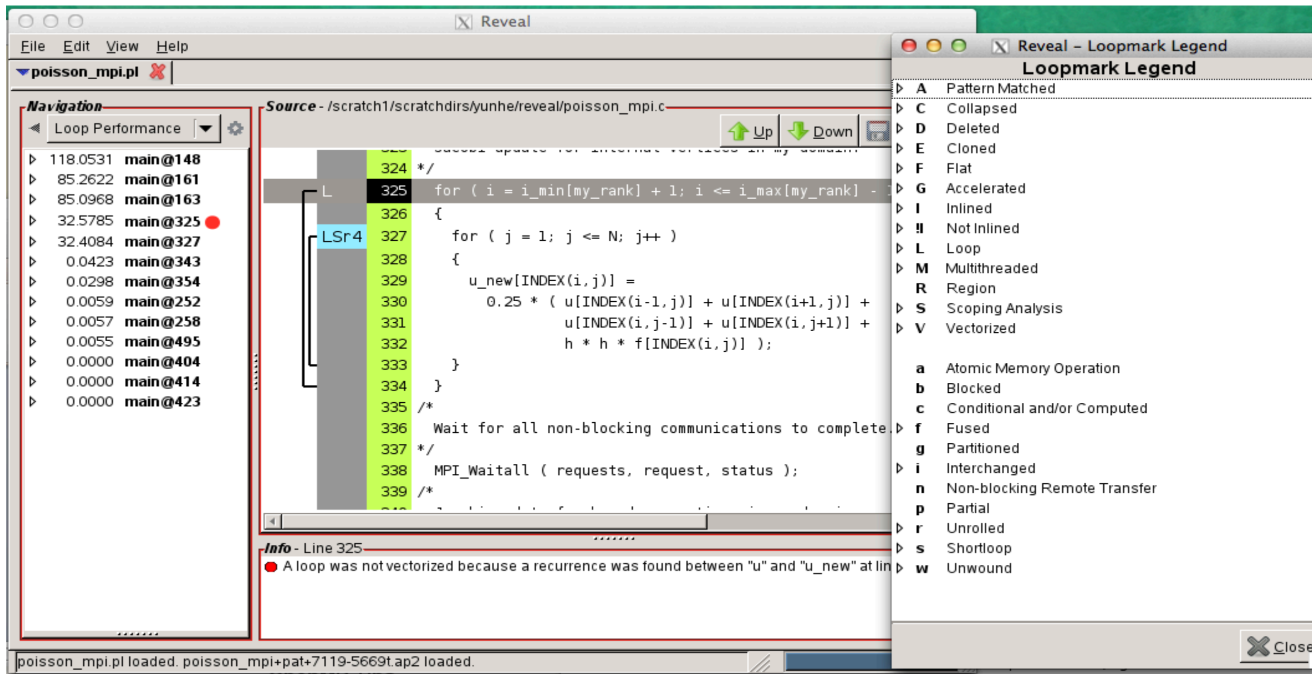
Insert Directive | Show Directive

Suggested OpenMP directives



- Make sure to always **save a copy of your original file first** before click the "Insert Directive" button. Your original file will be overwritten!!

Extensive “Help” topics in Reveal



OpenMP Tips

Reduction in an inlined function

There is a reduction to a variable which is in a called function.
The OMP programming model does not provide any method for automatically protecting the reduction variable.
For the loop to run correctly, the user needs to protect this reduction with a lock or change it to an atomic operation.

Note:
Because of current limitations in the analysis, if a global variable is inlined anywhere into the calling function, it must assume there is an inlined referenc in any loop which has any reference to the variable.
As a consequence of this, Reveal may display this scoping problem when it does not really exist.

Note:
Since the addition of a lock or changing to an atomic operation may significantly decrease performance, it may to necessary to clone the function containing the reduction if it is called from other places where the protection is unnecessary.

on a loop and select "Scope Loop" to make it a candidate for scoping.
You can select multiple loops before starting the scoping. A new window titled "Reveal OpenMP Scoping" will appear. When ready to start the scoping, select "Start Scoping" at the bottom left of the "Scope Loops" tab of the Scoping window. Files listed in the navigation panel will have a red or green icon added when they contain scoping information.

When loop performance information is attached, this enables performance filtering in the Scoping window. One way to use this feature is to add all the the program's loops using the "Edit List" menu in the scoping window and then using the filtering to uncheck loops that are not important. You may filter by execution time or number of loop trips. Setting a Time of 5.0 seconds will modify the loop list so that only loops using >= 5.0 seconds will be checked.

- **Create OpenMP clauses or directives**
- **View and Save OpenMP directives**
- **Compiler Message Navigation**
- **Notes**

Close

Reveal helps to start adding OpenMP

- **Only under PrgEnv-cray, with CCE compiler**
- **Start from most time consuming loops first**
- **Insert OpenMP directives**
  - Make sure to save a copy of the original code first, since the saved new file will overwrite the original code
- **There will be unresolved and incomplete variable scopes**
- **There maybe more incomplete and incorrect variables identified when compiling the resulted OpenMP codes**
- **User still needs to understand OpenMP, and resolves the issues.**
- **Verify correctness and performance**
- **Repeat as necessary**
- **No OpenMP tasks, barrier, critical, atomic regions, etc**

More work after reveal (1)

Reveal suggests:

```
#pragma omp parallel for default(none)                 \
      unresolved (my_change,my_n)                      \
      shared  (my_rank,N,i_max,u_new,u)                \
      firstprivate (i)

   for ( i = i_min[my_rank]; i <= i_max[my_rank]; i++ )
   {
    for ( j = 1; j <= N; j++ )
    {
     if ( u_new[INDEX(i,j)] != 0.0 )
     {
      my_change = my_change
       + fabs ( 1.0 - u[INDEX(i,j)] / u_new[INDEX(i,j)] );

      my_n = my_n + 1;
     }
    }
   }
```

Final code:

```
#pragma omp parallel for default(none)                 \
      private (my_change,my_n)                         \
      shared  (my_rank,N,i_min,i_max,u_new,u)          \
      private (j)                                      \
      private (i)

   for ( i = i_min[my_rank]; i <= i_max[my_rank]; i++ )
   {
    for ( j = 1; j <= N; j++ )
    {
     if ( u_new[INDEX(i,j)] != 0.0 )
     {
      my_change = my_change
       + fabs ( 1.0 - u[INDEX(i,j)] / u_new[INDEX(i,j)] );

      my_n = my_n + 1;
     }
    }
   }
```

More work after Reveal (2)

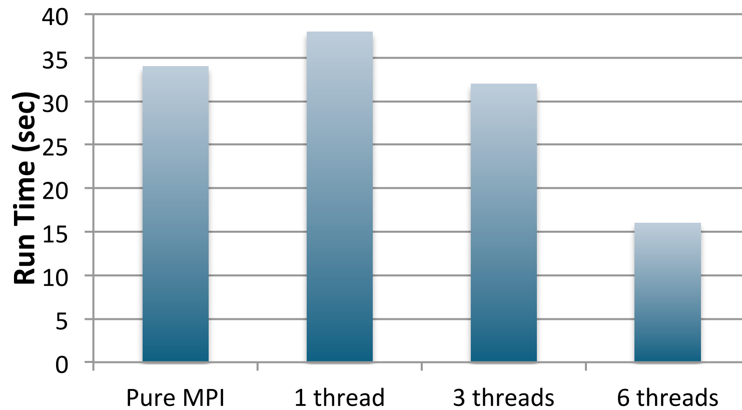
Reveal suggests:

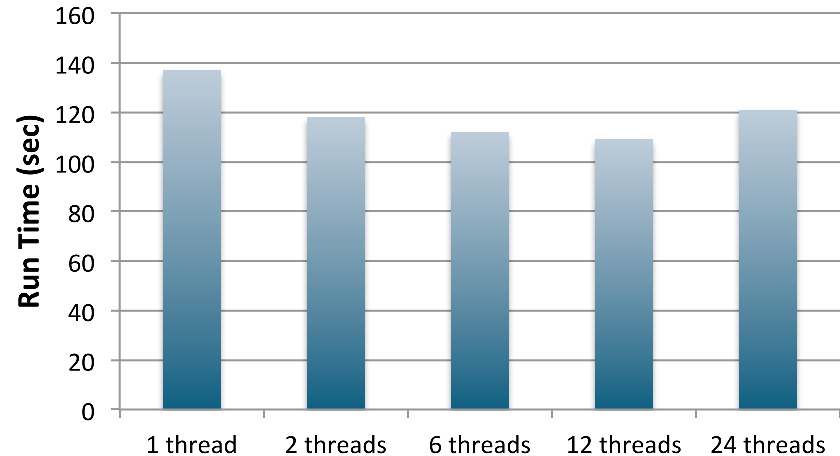
```
#pragma omp parallel for default(none)                \
      shared  (f,N,u,u_new,i,h)

 for ( i = i_min[my_rank] + 1; i <= i_max[my_rank] - 1; i++ )
 {
   for ( j = 1; j <= N; j++ )
   {
     u_new[INDEX(i,j)] =
       0.25 * ( u[INDEX(i-1,j)] + u[INDEX(i+1,j)] +
                u[INDEX(i,j-1)] + u[INDEX(i,j+1)] +
                h * h * f[INDEX(i,j)] );
   }
 }
```

Final code:

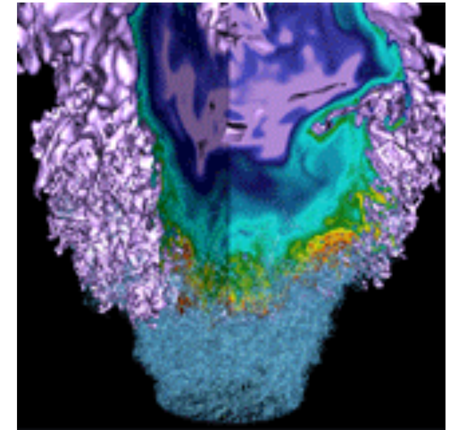
```
#pragma omp parallel for default(none)        \
      private (my_rank,j,i)                   \
      shared  (f,N,u,u_new,h,i_min,i_max)

 for ( i = i_min[my_rank] + 1; i <= i_max[my_rank] - 1; i++ )
 {
   for ( j = 1; j <= N; j++ )
   {
     u_new[INDEX(i,j)] =
       0.25 * ( u[INDEX(i-1,j)] + u[INDEX(i+1,j)] +
                u[INDEX(i,j-1)] + u[INDEX(i,j+1)] +
                h * h * f[INDEX(i,j)] );
   }
 }
```

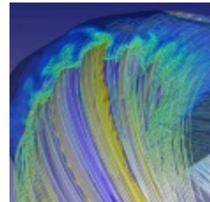
Performance with OpenMP added

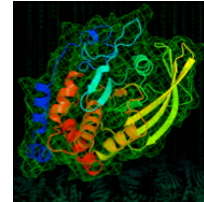


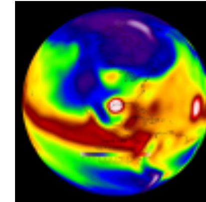


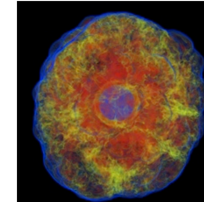
Hybrid MPI/OpenMP Scaling

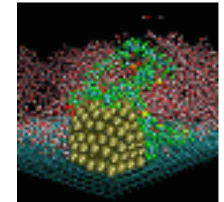












NERSC 40 YEARS at the FOREFRONT 1974-2014

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

The Big Picture

- **The next large NERSC production system "Cori" will be Intel Xeon Phi KNL (Knights Landing) architecture:**
  - >60 cores per node, 4 hardware threads per core
  - Total of >240 threads per node
- **Your application is very likely to run on KNL with simple port, but high performance is harder to achieve.**
- **Many applications will not fit into the memory of a KNL node using pure MPI across all HW cores and threads because of the memory overhead for each MPI task.**
- **Hybrid MPI/OpenMP is the recommended programming model, to achieve scaling capability and code portability.**
- **Current NERSC systems (Babbage, Edison, and Hopper) can help prepare your codes.**

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. Reduce memory footprint.**
- **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.**

Hybrid MPI/OpenMP Reduces Memory Usage

- **Smaller number of MPI processes. Save the memory needed for the executables and process stack copies.**
- **Larger domain for each MPI process, so fewer ghost cells**
  - e.g. Combine 16 10x10 domains to one 40x40. Assume 2 ghost layers.
  - Total grid size: Original: 16x14x14=3136, new: 44x44=1936.
- **Save memory for MPI buffers due to smaller number of MPI tasks.**
- **Fewer messages, larger message sizes, and smaller MPI all-to-all communication sizes improve performance.**

A Pseudo Hybrid Code

NERSC 40 YEARS at the FOREFRONT

```
Program hybrid
 call MPI_INIT (ierr)
 call MPI_COMM_RANK (…)
 call MPI_COMM_SIZE (…)
  …  some computation and MPI communication
  call OMP_SET_NUM_THREADS(4)
 !$OMP PARALLEL DO PRIVATE(i)
 !$OMP&                    SHARED(n)
    do i=1,n
        … computation
    enddo
  !$OMP END PARALLEL DO
  …  some computation and MPI communication
 call MPI_FINALIZE (ierr)
 end
```

MPI_INIT_Thread Choices

- **MPI_INIT_THREAD (required, provided, ierr)**
  - IN: required, desired level of thread support (integer).
  - OUT: provided, provided level of thread support (integer).
  - Returned provided maybe less than required.
- **Thread support levels:**
  - MPI_THREAD_SINGLE: Only one thread will execute.
  - MPI_THREAD_FUNNELED: Process may be multi-threaded, but only master thread will make MPI calls (all MPI calls are ''funneled'' to master thread)
  - MPI_THREAD_SERIALIZED: Process may be multi-threaded, multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are ''serialized'').
  - MPI_THREAD_MULTIPLE: Multiple threads may call MPI, with no restrictions.

Thread Support Levels

environment variable MPICH_MAX_THREAD_SAFETY	thread support level
not set	MPI_THREAD_SINGLE
single	MPI_THREAD_SINGLE
funneled	MPI_THREAD_FUNNELED
serialized	MPI_THREAD_SERIALIZED
multiple	MPI_THREAD_MULTIPLE

MPI Calls Inside OMP MASTER

- MPI_THREAD_FUNNELED is required.
- OMP_BARRIER is needed since there is no synchronization with OMP_MASTER.
- It implies all other threads are sleeping!

```
!$OMP BARRIER
!$OMP MASTER
    call MPI_xxx(...)
!$OMP END MASTER
!$OMP BARRIER
```

MPI Calls Inside OMP SINGLE

- MPI_THREAD_SERIALIZED is required.
- OMP_BARRIER is needed since OMP_SINGLE only guarantees synchronization at the end.
- It also implies all other threads are sleeping!

```
!$OMP BARRIER
!$OMP SINGLE
    call MPI_xxx(…)
!$OMP END SINGLE
```

THREAD FUNNELED/SERIALIZED vs. Pure MPI

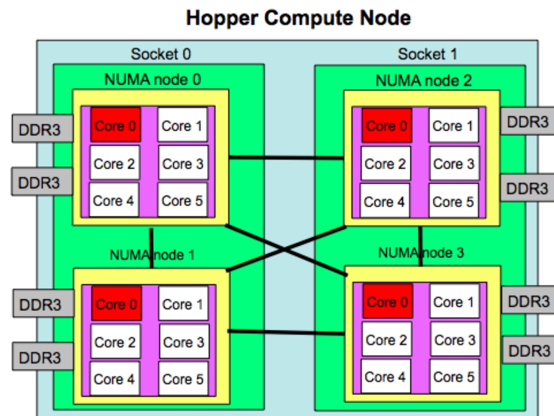
- **FUNNELED/SERIALIZED:**
  - All other threads are sleeping while single thread communicating.
  - Only one thread communicating maybe not able to saturate the inter-node bandwidth.
- **Pure MPI:**
  - Every CPU communicating may over saturate the inter-node bandwidth.
- **Overlap communication with computation!**

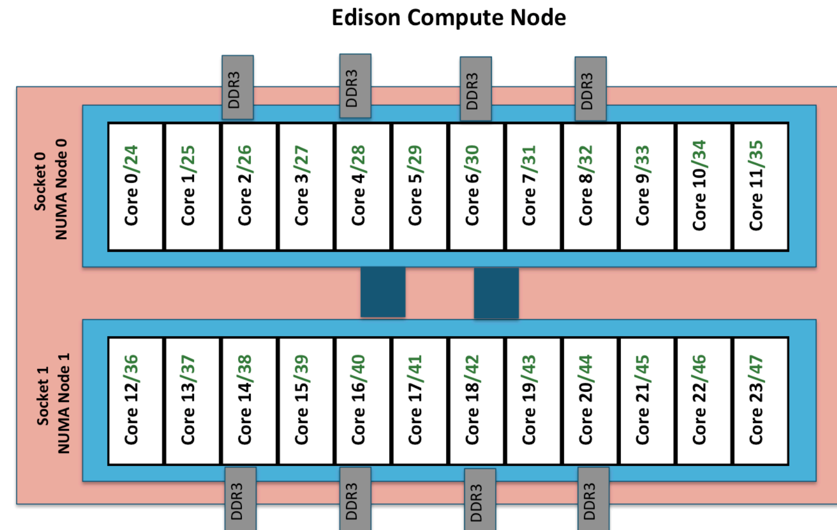
Overlap COMM and COMP

- Need at least MPI_THREAD_FUNNELED.
- Many “easy” hybrid programs only need MPI_THREAD_FUNNELED
- While master or single thread is making MPI calls, other threads are computing
- Must be able to separate codes that can run before or after halo info is received. Very hard
- Lose compiler optimizations.

```
!$OMP PARALLEL
    if (my_thread_rank < 1) then
        call MPI_xxx(…)
    else
        do some computation
    endif
!$OMP END PARALLEL
```

Hopper/Edison Compute Nodes



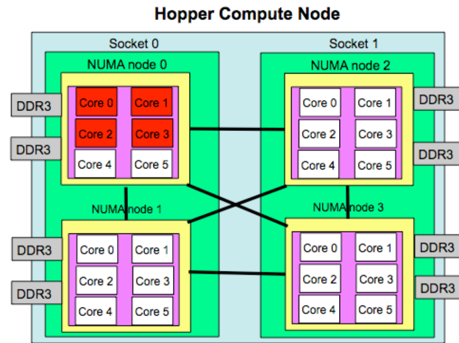


- **Hopper: NERSC Cray XE6, 6,384 nodes, 153,126 cores.**
  - **4 NUMA domains per node, 6 cores per NUMA domain.**
- **Edison: NERSC Cray XC30, 5,576 nodes, 133,824 cores.**
  - **2 NUMA domains per node, 12 cores per NUMA domain.
2 hardware threads per core.**
- **Memory bandwidth is non-homogeneous among NUMA domains.**

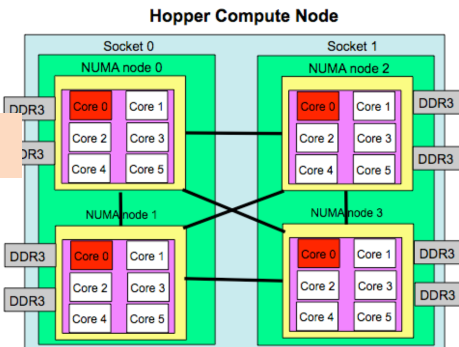
MPI Process Affinity: aprun "-S" Option

- **Process affinity: or CPU pinning, binds MPI process to a CPU or a ranges of CPUs on the node.**
- **Important to spread MPI ranks evenly onto different NUMA nodes.**
- **Use the "-S" option for Hopper/Edison.**

aprun –n 4 –d 6

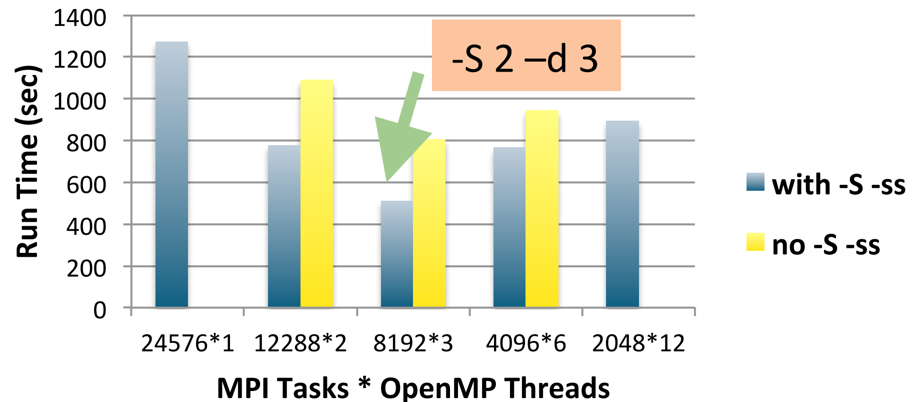


aprun –n 4 **–S 1**–d 6



Lower is Better

GTC Hybrid MPI/OpenMP on Hopper, 24,576 cores



Hopper Core Affinity

NERSC 40 YEARS at the FOREFRONT

- **"xthi.c": a hybrid MPI/OpenMP code that reports process and thread affinity.**
- **Source code can be found at (page 95-96): http://docs.cray.com/books/S-2496-4101/S-2496-4101.pdf**

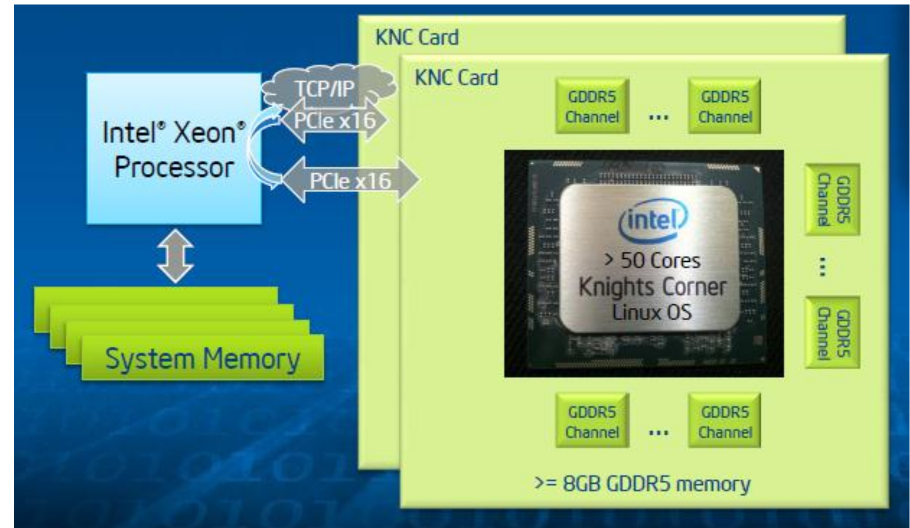
```
% aprun -n 4 ./xthi
Hello from rank 0, thread 0, on nid01085. (core affinity = 0)
Hello from rank 1, thread 0, on nid01085. (core affinity = 1)
Hello from rank 3, thread 0, on nid01085. (core affinity = 3)
Hello from rank 2, thread 0, on nid01085. (core affinity = 2)

% aprun -n 4 -S 1  ./xthi
Hello from rank 3, thread 0, on nid01085. (core affinity = 18)
Hello from rank 0, thread 0, on nid01085. (core affinity = 0)
Hello from rank 2, thread 0, on nid01085. (core affinity = 12)
Hello from rank 1, thread 0, on nid01085. (core affinity = 6)
```

Thread Affinity: aprun "-cc" Option

- **Thread affinity: forces each process or thread to run on a specific subset of processors, to take advantage of local process state.**
- **Thread locality is important since it impacts both memory and intra-node performance.**
- **On Hopper/Edison:**
  – The default option is "-cc cpu" (use it for non-Intel compilers)
  – Pay attention to Intel compiler, which uses an extra thread.
    - Use "-cc none" if 1 MPI process per node
    - Use "-cc numa_node" (Hopper) or "-cc depth" (Edison) if multiple MPI processes per node

Babbage

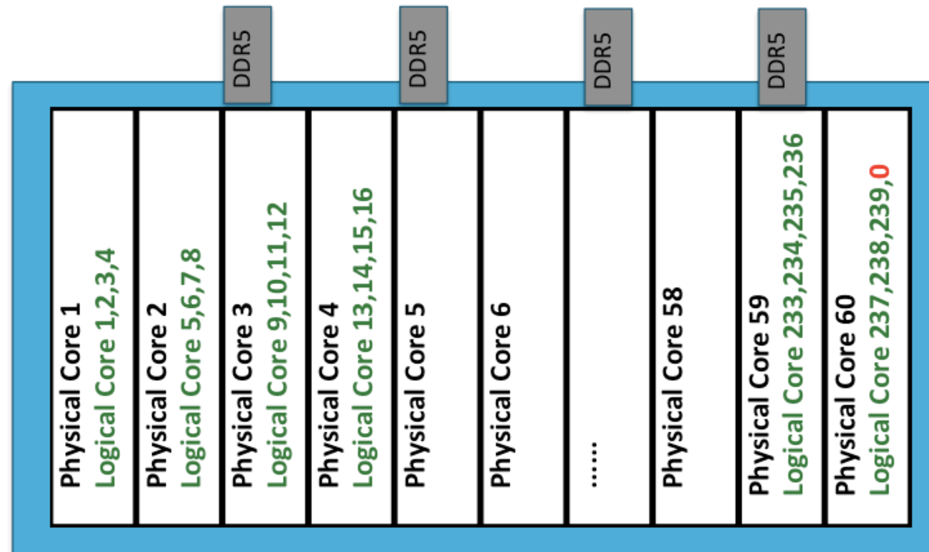
- **NERSC Intel Xeon Phi Knights Corner (KNC) testbed.**
- **45 compute nodes, each has:**
  - Host node: 2 Intel Xeon Sandybridge processors, 8 cores each.
  - 2 MIC cards each has 60 native cores and 4 hardware threads per core.
  - MIC cards attached to host nodes via PCI-express.
  - 8 GB memory on each MIC card
- **Recommend to use at least 2 threads per core to hide latency of in-order execution.**



To best prepare codes on Babbage for Cori:

- Use "native" mode on KNC to mimic KNL, which means ignore the host, just run completely on KNC cards.
- Encourage single node exploration on KNC cards with problem sizes that can fit.

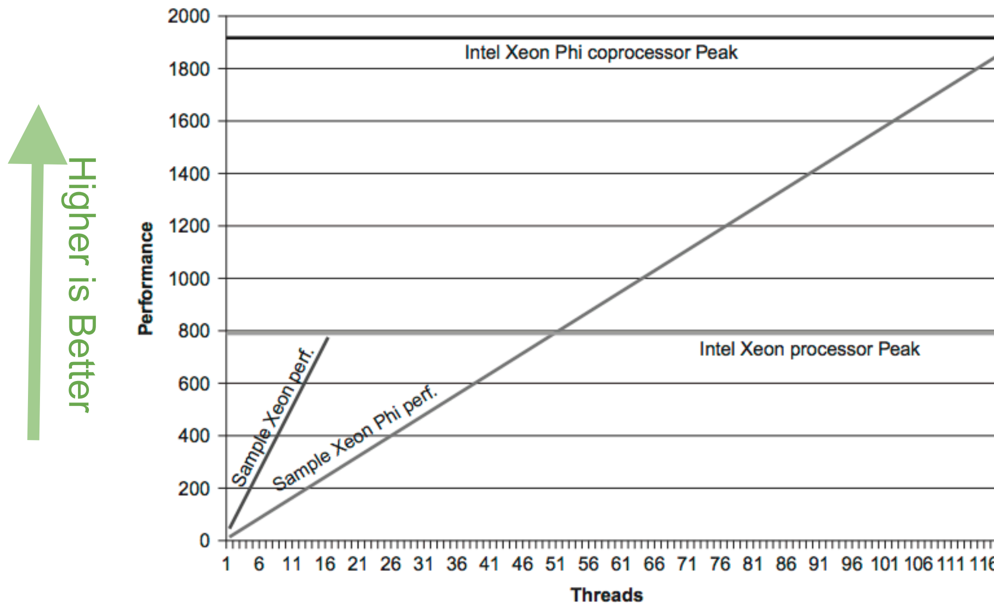
Babbage MIC Card



Babbage: NERSC Intel Xeon Phi testbed, 45 nodes.

- **1 NUMA domain per MIC card: 60 physical cores, 240 logical cores.**
- **Process affinity: spread MPI process onto different physical cores.**
- **Logical core 0 is on physical core 60.**

Why Scaling is So Important



Courtesy of Jim Jeffers and James Reinders

- **Scaling of an application is important to get the performance potential on the Xeon Phi manycore systems.**
- **Does not imply to scale with "pure MPI" or "pure OpenMP"**
- **Does not imply the need to scale all the way to 240-way either**
- **Rather, should explore hybrid MPI/OpenMP, find some sweet spots with combinations, such as: 4 MPI tasks * 15 threads per task, or 8*20, etc.**

Thread Affinity: KMP_AFFINTIY

- Run Time Environment Variable.
- none: no affinity setting. Default setting on the host.
- compact: default option on MIC. Bind threads as close to each other as possible

Node	Core 1				Core 2				Core 3			
	HT1	HT2	HT3	HT4	HT1	HT2	HT3	HT4	HT1	HT2	HT3	HT4
Thread	0	1	2	3	4	5						

- scatter: bind threads as far apart as possible. Default setting on MIC.

Node	Core 1				Core 2				Core 3			
	HT1	HT2	HT3	HT4	HT1	HT2	HT3	HT4	HT1	HT2	HT3	HT4
Thread	0	3			1	4			2	5		

- balanced: only available on MIC. Spread to each core first, then set thread numbers using different HT of same core close to each other.

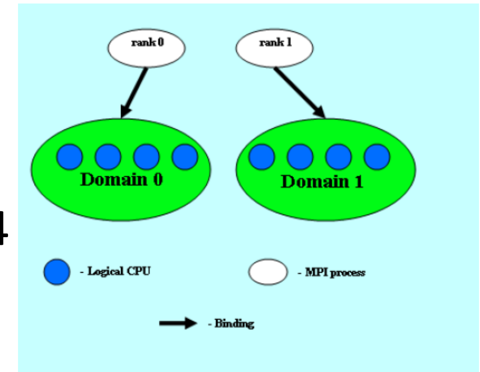
Node	Core 1				Core 2				Core 3			
	HT1	HT2	HT3	HT4	HT1	HT2	HT3	HT4	HT1	HT2	HT3	HT4
Thread	0	1			2	3			4	5		

- explicit: example: setenv KMP_AFFINITY "explicit, granularity=fine, proclist=[1:236:1]"
- New env on coprocessors: KMP_PLACE_THREADS, for exact thread placement

Thread Affinity: KMP_PLACE_THREADS

- **New setting on MIC only**. In addition to KMP_AFFINITY, can set exact but still generic thread placement.
- **KMP_PLACE_THREADS**=<n>Cx<m>T,<o>O
  - <n> Cores times <m> Threads with <o> of cores Offset
  - e.g. 40Cx3T,1O means using 40 cores, and 3 threads (HT2,3,4) per core
- **OS runs on logical proc 0, which lives on physical core 60**
  - OS procs on core 60: 0,237,238,239.
  - Avoid use proc 0

MPI Process Affinity: I_MPI_PIN_DOMAIN



- **A domain is a group of logical cores**
  - Domains are non-overlapping
  - Number of logical cores per domain is a multiple of 4
  - 1 MPI process per domain
  - OpenMP threads can be pinned inside each domain
- **I_MPI_PIN_DOMAIN=<size>[:<layout>]**

<size> = omp — adjust to OMP_NUM_THREADS
auto — #CPUs/ #MPI procs
<n> — a number

<layout> = platform — according to BIOS numbering
compact — close to each other
scatter — far away from each other

Hybrid Parallelization Strategies

- From sequential code, decompose with MPI first, then add OpenMP.
- From OpenMP code, treat as serial code.
- From MPI code, add OpenMP.
- Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks. MPI_THREAD_FUNNELED is usually the best choice.
- Could use MPI inside parallel region with thread-safe MPI.
- Avoid MPI_THREAD_MULTIPLE if you can. It slows down performance due to the usage of global locks for thread safety.

Programming Tips for Adding OpenMP

- **Choose between fine grain or coarse grain parallelism implementation.**
- **Use profiling tools to find hotspots. Add OpenMP and check correctness incrementally.**
- **Parallelize outer loop and collapse loops if possible.**
- **Minimize shared variables, minimize barriers.**
- **Decide whether to overlap MPI communication with thread computation.**
  - Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks.
  - Could use MPI inside parallel region with thread-safe MPI.
- **Consider OpenMP TASK.**

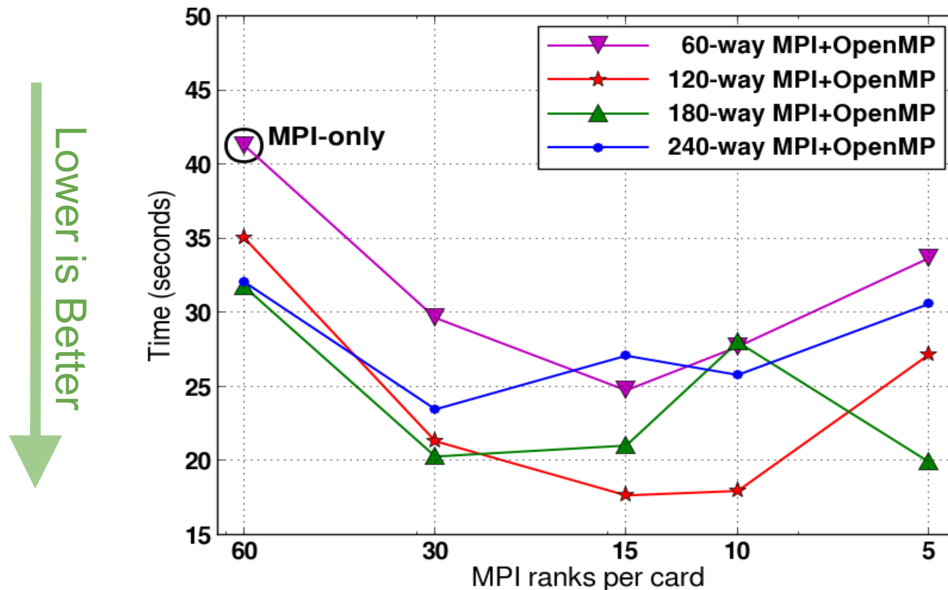
Why Hybrid MPI/OpenMP Code is Sometimes Slower?

- **All threads are idle except one while MPI communication.**
  - Need overlap comp and comm for better performance.
  - Critical Section for shared variables.
- **Thread creation overhead**
- **Cache coherence, false sharing.**
- **Data placement, NUMA effects.**
- **Natural one level parallelism problems.**
- **Pure OpenMP code performs worse than pure MPI within node.**
- **Lack of optimized OpenMP compilers/libraries.**

If a Routine Does Not Scale Well

- Examine code for serial/critical sections, eliminate if possible.
- Reduce number of OpenMP parallel regions to reduce overhead costs.
- Perhaps loop collapse, loop fusion or loop permutation is required to give all threads enough work, and to optimize thread cache locality. Use NOWAIT clause if possible.
- Pay attention to load imbalance. If needed, try dynamic scheduling or implement own load balance scheme.
- Experiment with different combinations of MPI tasks and number of threads per task. Less MPI tasks may not saturate inter-node bandwidth.
- Test different process and thread affinity options.
- Leave some cores idle on purpose, for memory capacity or bandwidth capacity.

MPI *vs.* OpenMP Scaling Analysis



Courtesy of Chris Daley, NERSC

- Each line represents multiple runs using fixed total number of cores = #MPI tasks x #OpenMP threads/task.
- Scaling may depend on the kernel algorithms and problem sizes.
- In this test case, 15 MPI tasks with 8 OpenMP threads per task is optimal.

- **Understand your code by creating the MPI *vs.* OpenMP scaling plot, find the sweet spot for hybrid MPI/OpenMP.**
- **It can be the base setup for further tuning and optimizing on Xeon Phi.**

Performance Analysis And Debugging

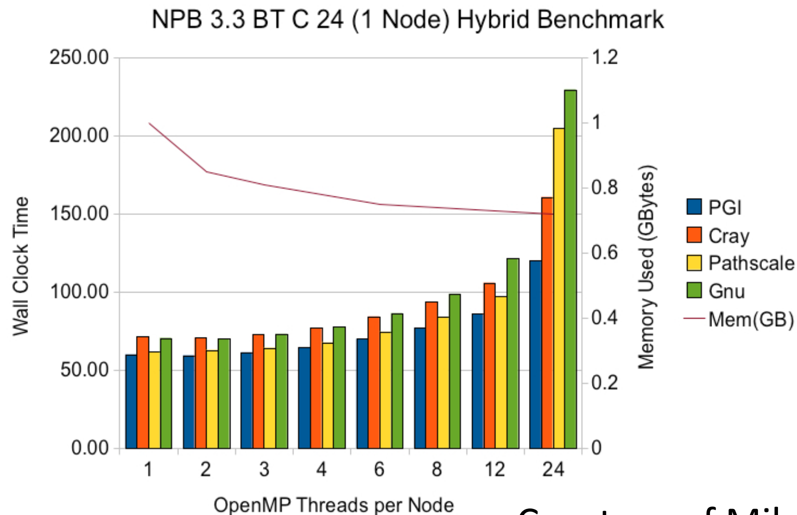
- **Performance Analysis**
  - Hopper/Edison:
    - Cray Performance Tools
    - IPM
    - Allinea MAP, perf-reports
    - TAU
  - Babbage:
    - Vtune
    - Intel Trace Analyzer and Collector
    - HPCToolkit
    - Allinea MAP
- **Debugging**
  - Hopper/Edison: DDT, Totalview, LGDB, Valgrind
  - Babbage: Intel Inspector, GDB, DDT

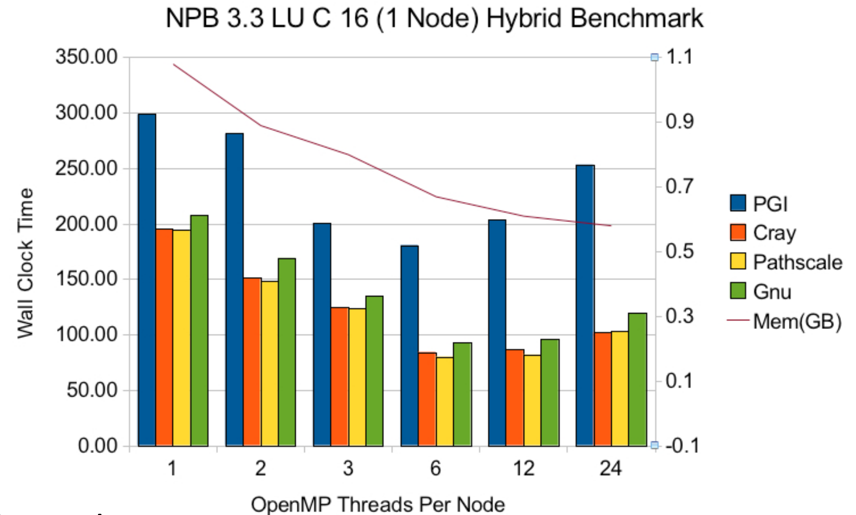
Case Studies Introduction

- **OpenMP parallelizing techniques used in real codes.**
- **Hybrid MPI/OpenMP applications on Hopper**
- **LBM on TACC Stampede** *(by Carlos Rosales,TACC)*
  - Add OpenMP incrementally
  - Compare OpenMP affinity
- **MFDn on Hopper** *(by H. Metin Aktulga et al., LBNL)*
  - Overlap communication and computation
- **NWChem on Babbage** *(by Hongzhang Shan et al., LBNL)*
  - CCSD(T)
    - Add OpenMP at the outermost loop level
    - Loop permutation, collapse
    - Reduction, remove loop dependency
  - Fock Matrix Construction (FMC)
    - Add OpenMP to most time consuming functions
    - OpenMP Task
    - Find sweet scaling spot with hybrid MPI/OpenMP

Hopper: Hybrid MPI/OpenMP NPB

NERSC 40 YEARS at the FOREFRONT





Courtesy of Mike Stewart

On a single node, hybrid MPI/OpenMP NAS Parallel Benchmarks:

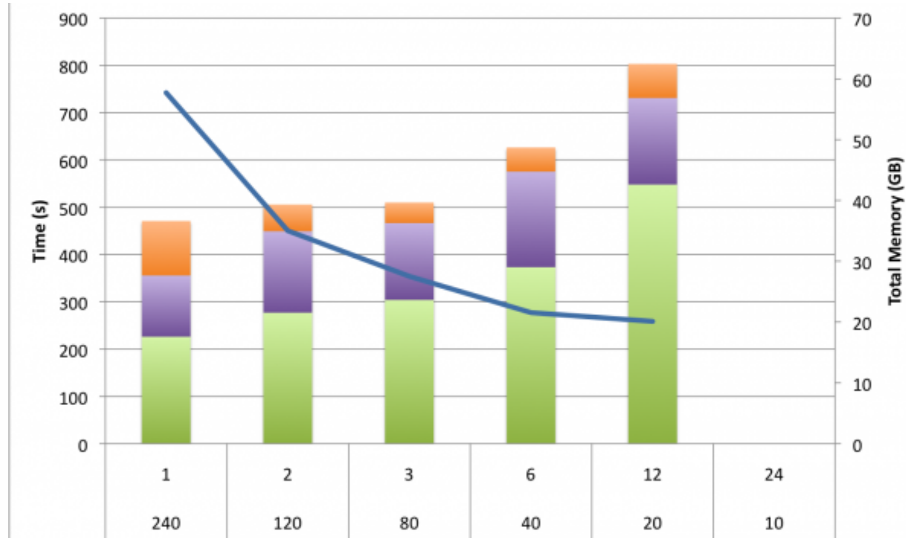
- **Reduced memory footprint with increased OpenMP threads.**
- **Hybrid MPI/OpenMP can be faster or comparable to pure MPI.**
- **Try different compilers.**
- **Sweet spot: BT: 1-3 threads; LU: 6 threads.**

U.S. DEPARTMENT OF ENERGY | Office of Science

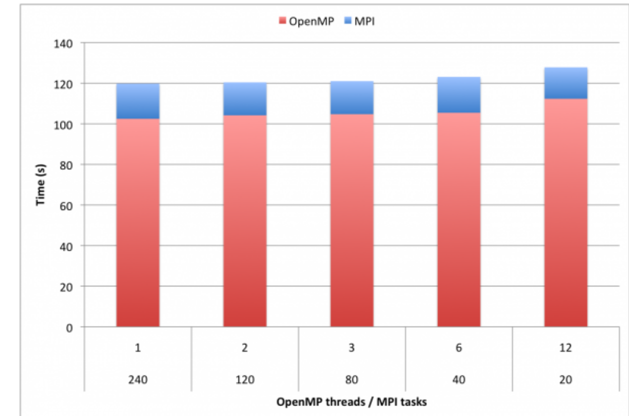
BERKELEY LAB

Hopper: Hybrid MPI/OpenMP fvCAM

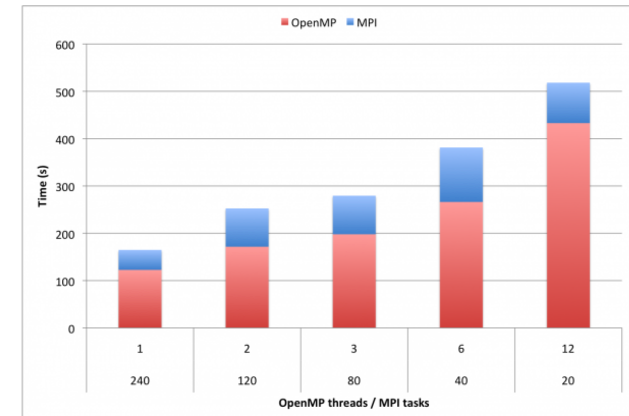
Total



"Physics" Component



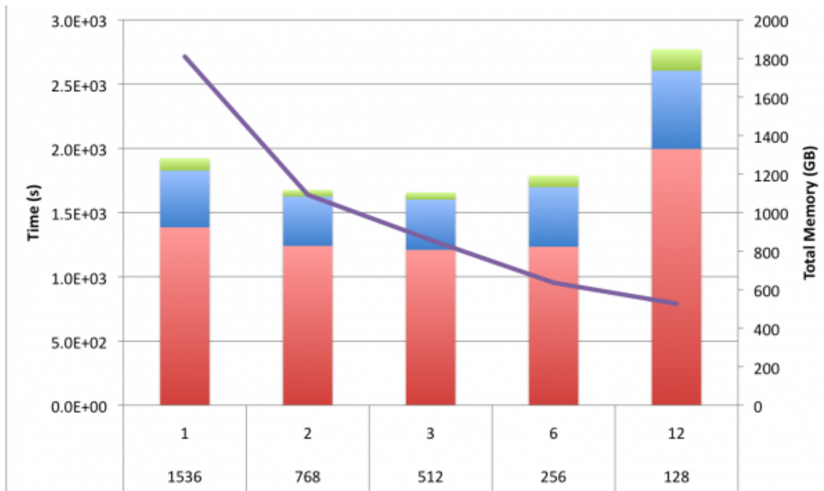
"Dynamics" Component

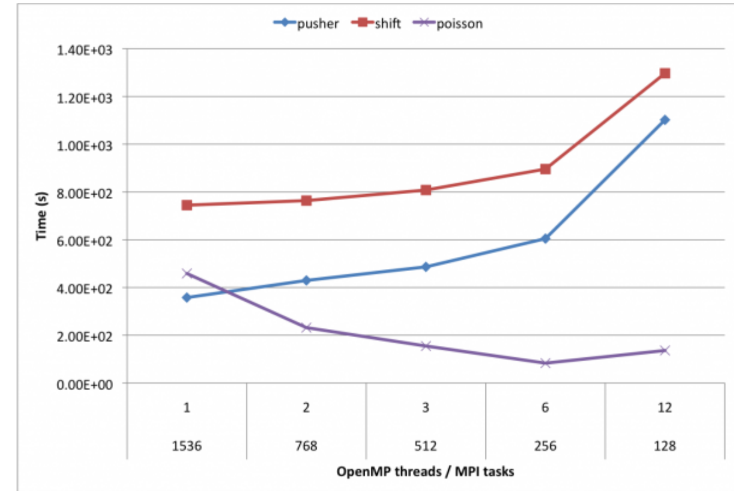


Community Atmospheric Model:
- **Memory reduces to 50% with 3 threads but only 6% performance drop.**
- **OpenMP time starts to grow from 6 threads.**
- **Load imbalance in "Dynamics" OpenMP**

Courtesy of Nick Wright, et. al, NERSC/Cray Center of Excellence"

Hopper: Hybrid MPI/OpenMP GTC





3d Gyrokinetic Toroidal Code:

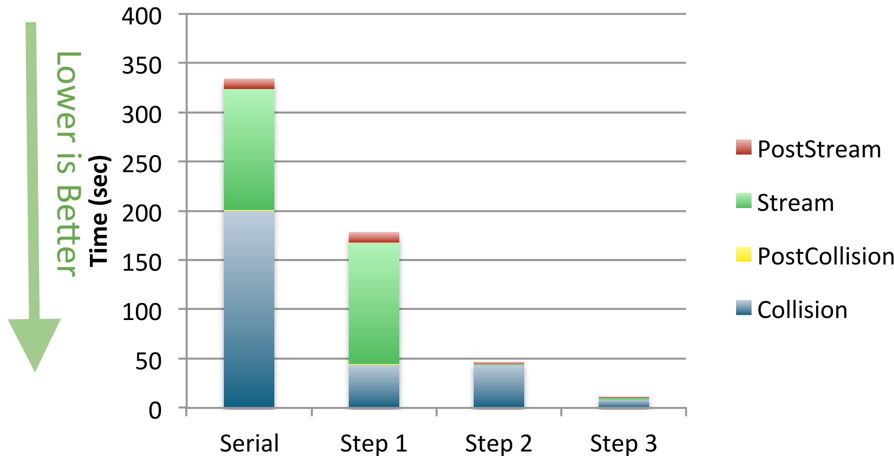
- **Memory reduces to 50% with 3 threads, also 15% better performance**
- **NUMA effects seen with 12 threads**
- **Mixed results in different kernels**

Courtesy of Nick Wright, et. al, NERSC/Cray Center of Excellence

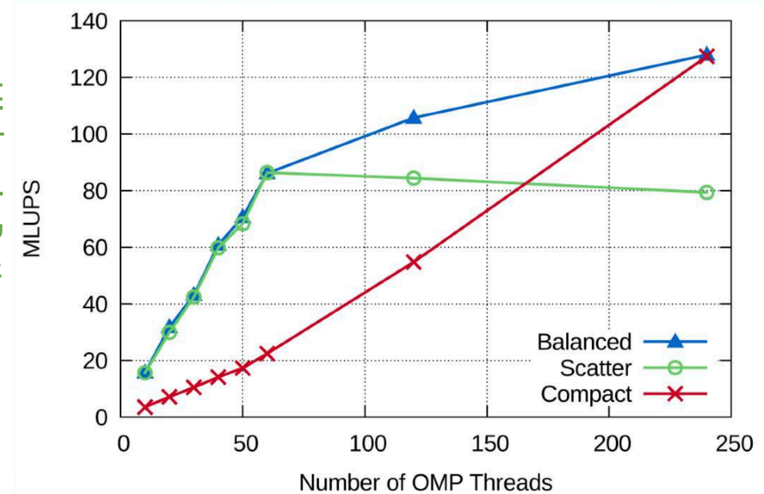
LBM, Add OpenMP Incrementally

NERSC 40 YEARS at the FOREFRONT

Lower is Better



Higher is Better

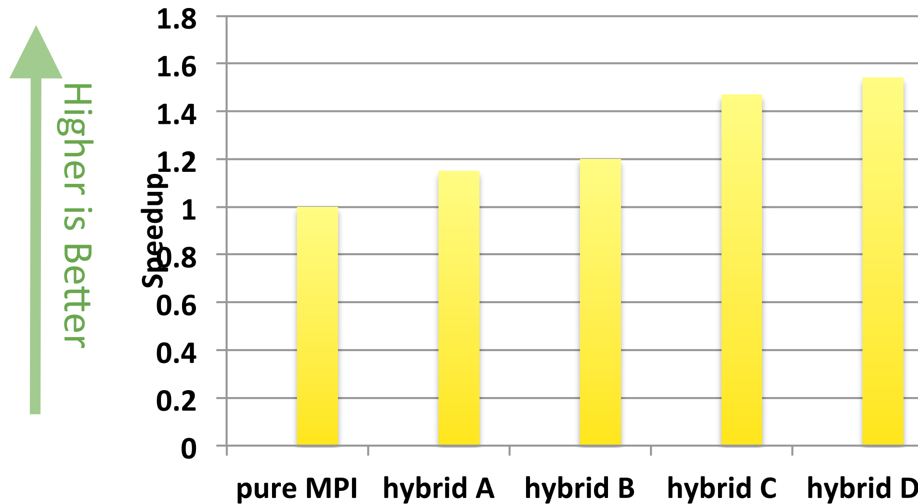


- **Lattice Boltzmann Method: a Computational Fluid Dynamics Code.**
- **Actual serial run time for Collision > 2500 sec** ***(plotted above as 200 sec only for better display)*****, > 95% of total run time.**
- **Step 1: Add OpenMP to hotspot Collision. 60X Collision speedup.**
- **Step 2: Add OpenMP to the new bottleneck, Stream and others. 89X Stream speedup.**
- **Step 3: Add vectorization. 5X Collision speedup.**
- **Balanced provides best performance overall.**

U.S. DEPARTMENT OF ENERGY | Office of Science

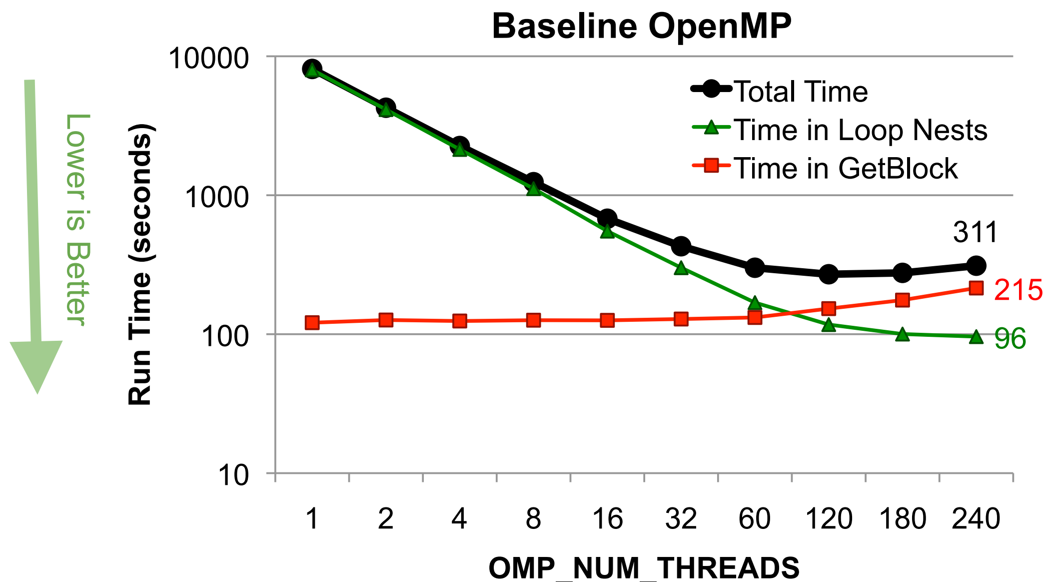
BERKELEY LAB

MFDn, Overlap Comm and Comp



- **MFDn: a nuclear physics code.**
- **Hopper. Pure MPI: 12,096 MPI tasks.**
- **Hybrid A: hybrid MPI/OpenMP, 2016 MPI* 6 threads.**
- **Hybrid B: hybrid A, plus: merge MPI_Reduce and MPI_Scatter into MPI_Reduce_Scatter, and merge MPI_Gather and MPI_Bcast into MPI_Allgatherv.**
- **Hybrid C: Hybrid B, plus: overlap row-group communications with computation.**
- **Hybrid D: Hybrid C, plus: overlap (most) column-group communications with computation.**

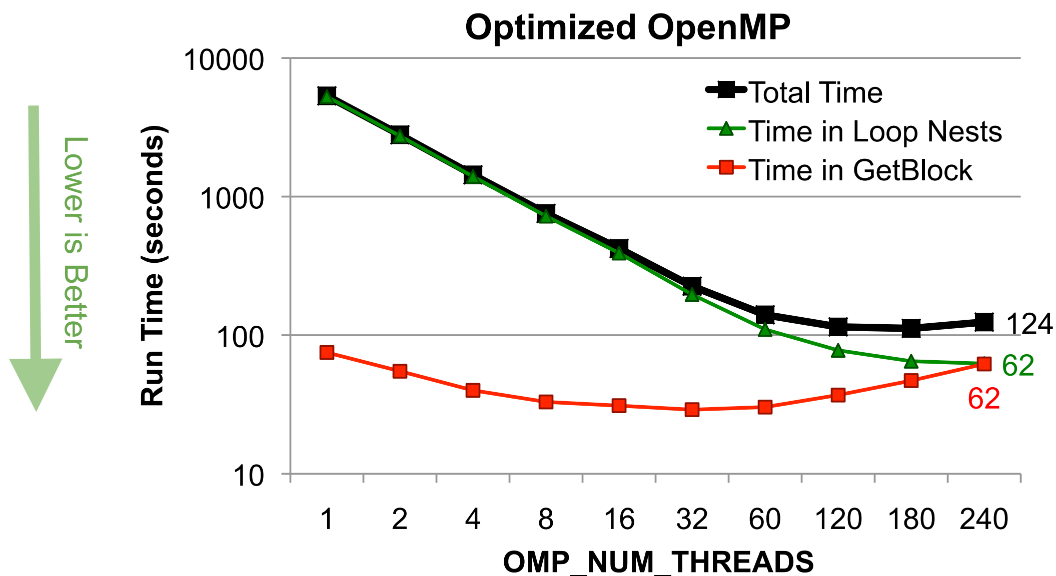
NWChem CCSD(T), Baseline OpenMP



- **Due to memory limitation, can only run with 1 MPI process per MIC.**
- **OpenMP added at the outermost loops of hotspots: Loop Nests. Scales well up to 120 threads.**
- **GetBlock is not parallelized with OpenMP. Hyper-threading hurts performance.**
- **Total time has perfect scaling from 1 to 16 threads. Best time at 120 threads.**
- **Balanced affinity gives best performance.**

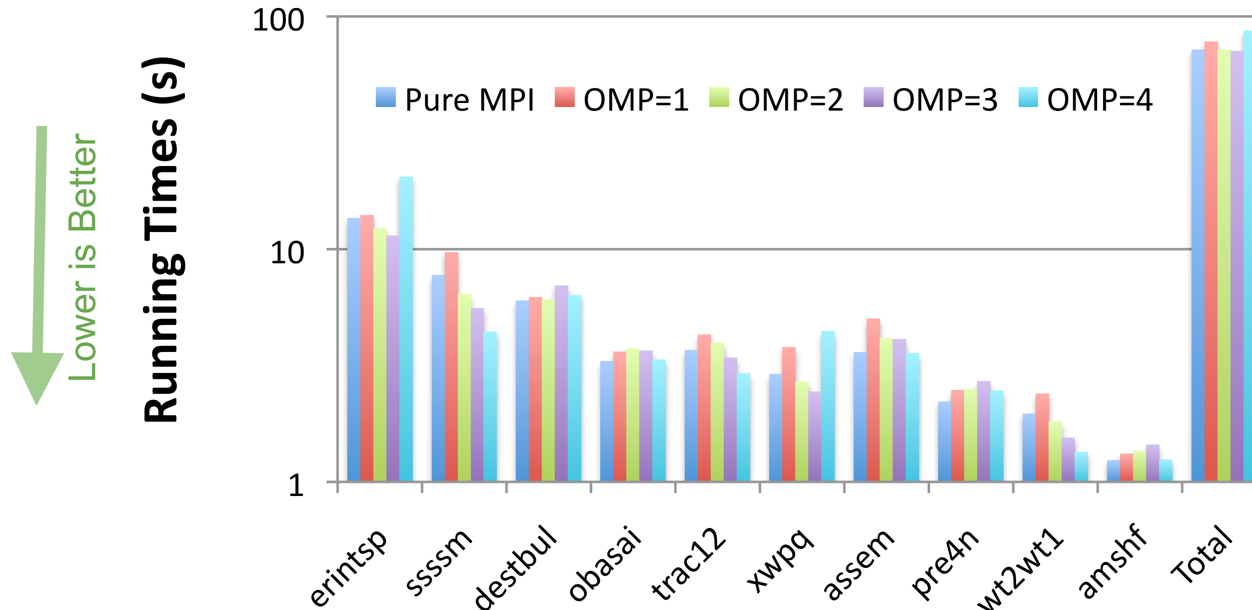
U.S. DEPARTMENT OF ENERGY | Office of Science
BERKELEY LAB

NWChem CCSD(T), More OpenMP Optimizations



- GetBlock optimizations: parallelize sort, loop unrolling.
- Reorder array indices to match loop indices.
- Merge adjacent loop indices to increase number of iterations.
- Align arrays to 64 bytes boundary.
- Exploit OpenMP loop control directive, provide complier hints.
- Total speedup from base is 2.3x.

NWChem FMC, Add OpenMP to HotSpots (OpenMP #1)



- Total number of MPI ranks=60; OMP=N means N threads per MPI rank.
- Original code uses a shared global task counter to deal with dynamic load balancing with MPI ranks
- Loop parallelize top 10 routines in TEXAS package (75% of total CPU time) with OpenMP. Has load-imbalance.
- OMP=1 has overhead over pure MPI.
- OMP=2 has overall best performance in many routines.

NWChem FMC, OpenMP Task Implementation (OpenMP #3)

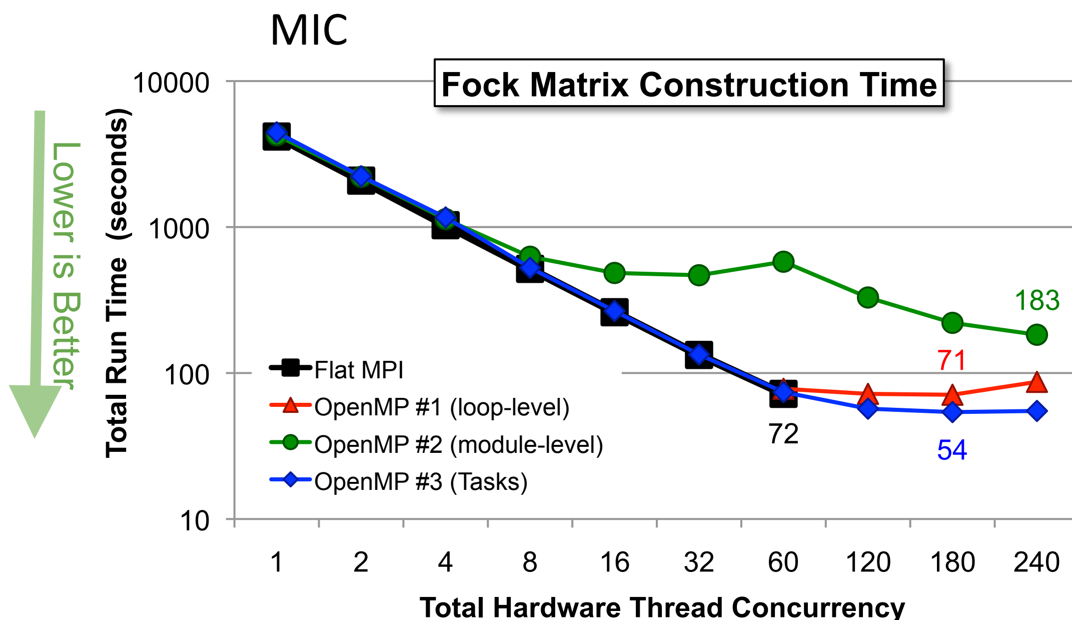
Fock Matrix Construction — OpenMP Task Implementation

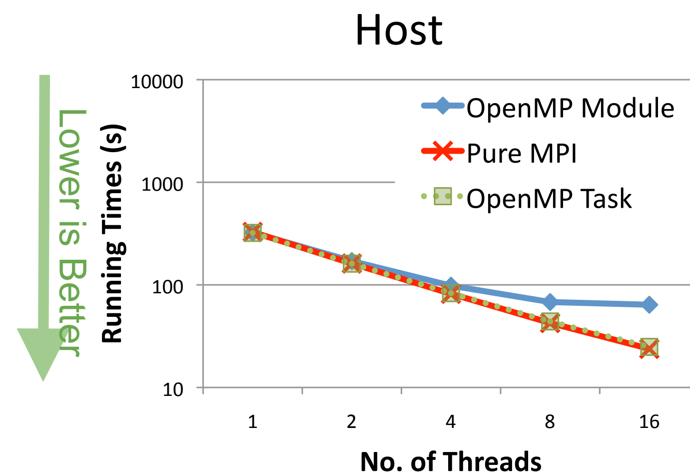
```
c$OMP parallel
myfock() = 0
c$OMP master
current_task_id = 0
mytid = omp_get_thread_num()
My_task = global_task_counter(task_block_size)
for ijkl = 2*ntype to 2 step −1 do
    for ij = min(ntype, ijkl − 1) to max(1, ijkl − ntype) step −1 do
        kl = ijkl − ij
        if (my_task .eq. current_task_id) then
          c$OMP task firstprivate(ij,kl) default(shared)
          create_task(ij,kl, ...)
          c$OMP end task
          my_task=global_task_counter(task_block_size)
        end if
        current_task_id = current_task_id + 1
    end for
end for
c$OMP end master
c$OMP taskwait
c$OMP end parallel
Perform Reduction on myfock to Fock matrix
```

- OpenMP task model is flexible and powerful.
- The task directive defines an explicit task.
- Threads share work from all tasks in the task pool.
- Master thread creates tasks.
- The taskwait directive makes sure all child tasks created for the current task finish.
- Helps to improve load balance.

- **Use OpenMP tasks.**
- **To avoid two threads updating Fock matrix simultaneously, a local copy is used per thread. Reduction at the end.**

NWChem FMC, Run Time

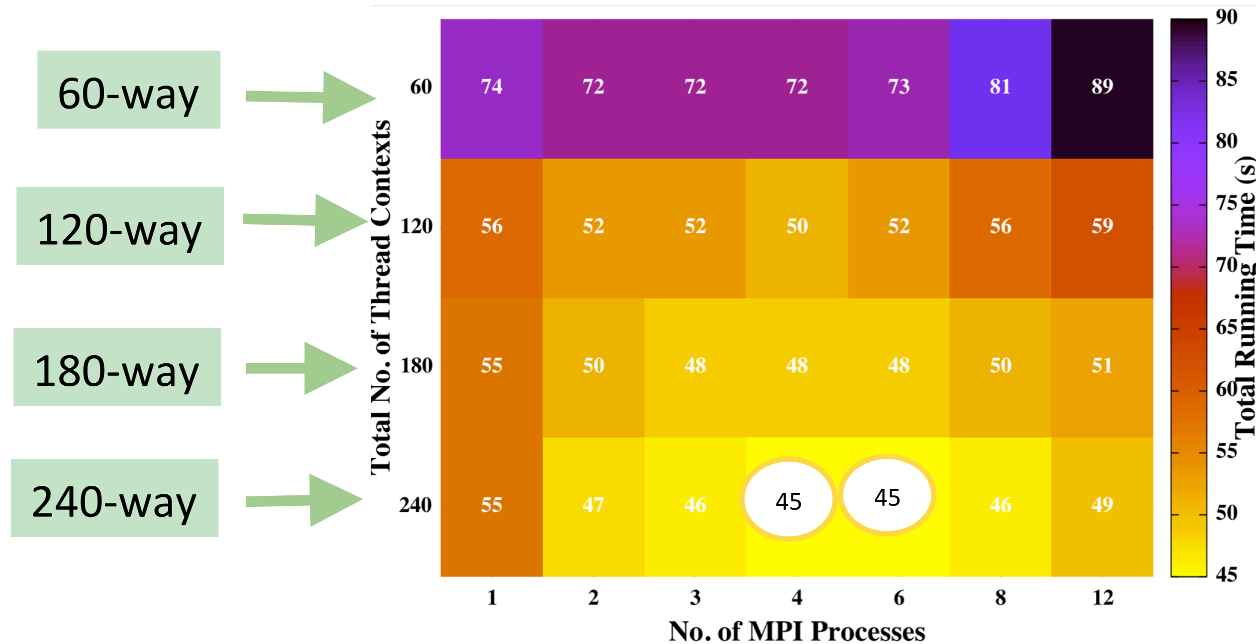




- **Flat MPI is limited to a total of 60 ranks due to memory limitation.**
- **OpenMP #1 uses flat MPI up to 60 MPI processes, then uses 2, 3, and 4 threads per MPI rank.**
- **OpenMP #2 and #3 are pure OpenMP.**
- **OpenMP #2 module-level parallelism saturates at 8 threads (critical and reduction related). Then when over 60 threads, hyper-threading helps.**
- **OpenMP #3 Task implementation continues to scale over 60 cores. 1.33x faster (with 180 threads) than pure MPI.**
- **The OpenMP Task implementation benefits both MIC and Host.**

NWChem FMC, MPI/OpenMP Scaling and Tuning

NERSC 40 YEARS at the FOREFRONT



- **Another way of showing scaling analysis result.**
- **Sweet spot is either 4 MPI tasks with 60 OpenMP threads per task, or 6 MPI tasks with 40 OpenMP threads per task.**
- **1.64x faster than original flat MPI.**
- **22% faster than 60 MPI tasks with 4 OpenMP threads per task.**

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Summary (1)

- OpenMP is a fun and powerful language for shared memory programming.
- Hybrid MPI/OpenMP is recommended for many next generation architectures (Intel Xeon Phi for example), including NERSC-8 system, Cori.
- You should explore to add OpenMP now if your application is flat MPI only.

Summary (2)

- **Use Edison/Babbage to help you to prepare for Cori regarding thread scalability.**
  - MPI performance across nodes or MIC cards on Babbage is not optimal.
  - Concentrate on optimization on single MIC card.
- **Case studies showed effectiveness of OpenMP**
  - Add OpenMP incrementally. Conquer one hotspot at a time.
  - Experiment with thread affinity choices. Balanced is optimal for most applications. Low hanging fruit.
  - Pay attention to cache locality and load balancing. Adopt loop collapse, loop permutation, etc.
  - Find sweet spot with MPI/OpenMP scaling analysis.
  - Consider OpenMP TASK. Major code rewrite.
  - Consider overlap communication with computation. Very hard to do.
- **Optimizations targeted for one architecture (XE6, XC30, KNC) can help performance for other architectures (Xeon, XC30, KNL).**

References

- OpenMP: http://openmp.org
- NERSC Hopper/Edison/Babbage web pages:
  - https://www.nersc.gov/users/computational-systems/hopper
  - https://www.nersc.gov/users/computational-systems/edison
  - https://www.nersc.gov/users/computational-systems/testbeds/babbage
- OpenMP Resources:
  - https://www.nersc.gov/users/computational-systems/edison/programming/using-openmp/openmp-resources/
- Improving OpenMP Scaling (online soon)
  - https://www.nersc.gov/users/computational-systems/cori/preparing-for-cori/improving-openmp-scaling/
- Cray Reveal at NERSC:
  - https://www.nersc.gov/users/training/events/cray-reveal-tool-training-sept-18-2014/
  - https://www.nersc.gov/users/software/debugging-and-profiling/craypat/reveal/
- H. M. Aktulga, C. Yang, E. G. Ng, P. Maris and J. P. Vary, "Improving the Scalability of a symmetric iterative eigensolver for multi-core platforms," Concurrency and Computation: Practice and Experience 25 (2013).
- Carlos Rosale, "Porting to the Intel Xeon Phi TACC paper: Opportunities and Challenges". Extreme Scaling Workshop 2013 (XSCALE2013), Boulder, CO, 2013.
- Hongzhang Shan, Samuel Williams, Wibe de Jong, Leonid Oliker, "Thread-Level Parallelization and Optimization of NWChem for the Intel MIC Architecture", LBNL Technical Report, October 2014, LBNL 6806E.
- Jim Jeffers and James Reinders, "Intel Xeon Phi Coprocessor High-Performance Programming". Published by Elsevier Inc. 2013.
- Intel Xeon Phi Coprocessor Developer Zone:
  - http://software.intel.com/mic-developer
- Interoperability with OpenMP API
  - http://software.intel.com/sites/products/documentation/hpc/ics/impi/41/win/Reference_Manual/Interoperability_with_OpenMP.htm