How to Thread a Hydro Code with OpenMP

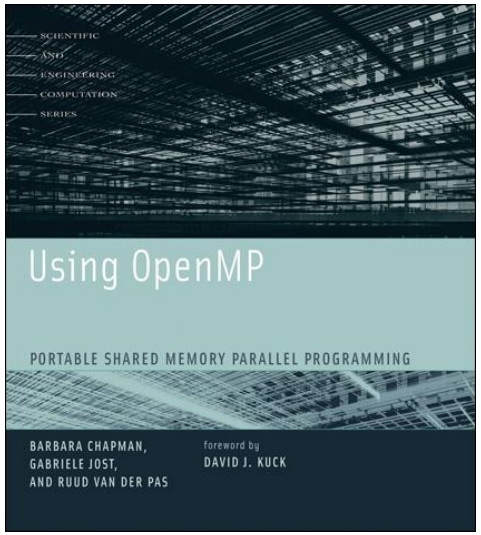
Andy Nonaka, Mike Lijewski
Center for Computational Sciences and Engineering
Lawrence Berkeley Laboratory
HIPACC Summer School – July 25, 2011

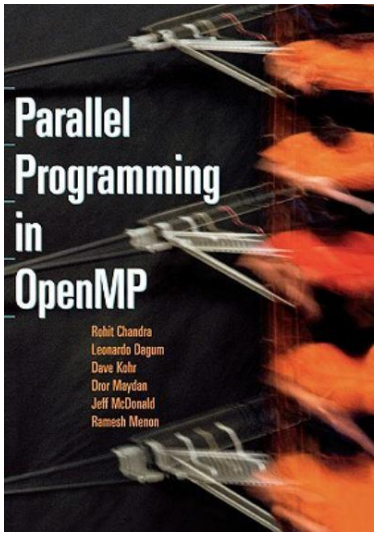
References

- OpenMP home page: www.openmp.org
- OpenMP user's group: www.compunity.org

- Books
  - "Using OpenMP", Chapman
  - "Parallel Programming in OpenMP", Chandra

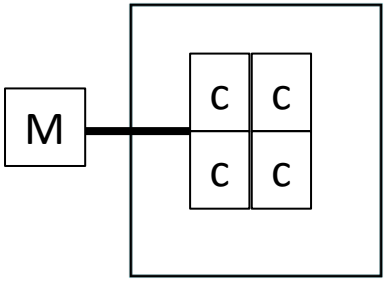
- NERSC (or other supercomputing center) web pages
- Web in general (e.g., google)



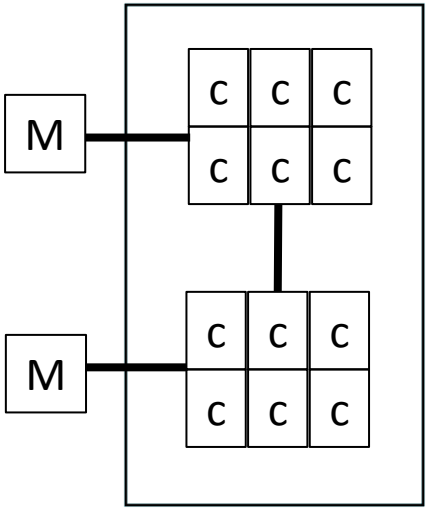


What is OpenMP?

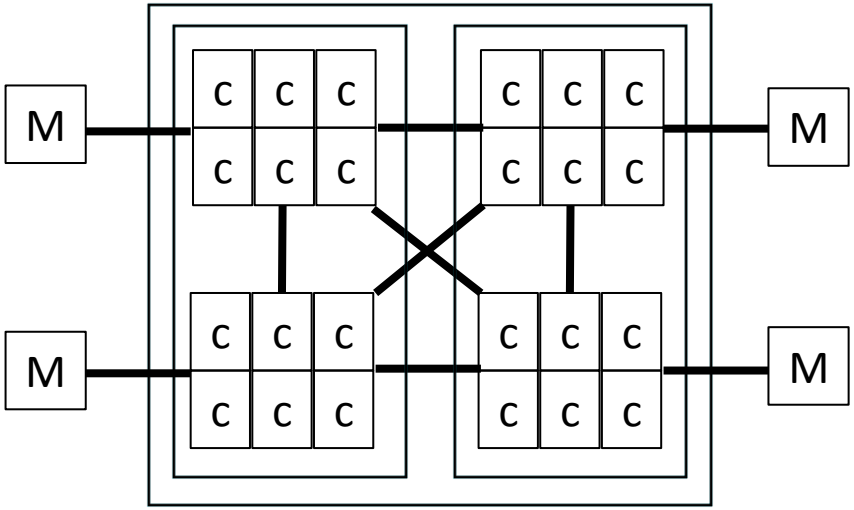
- OpenMP is a parallel programming model for multicore architectures with shared memory.
- In this talk, a “node” is some processing unit where all the cores can access the same memory without parallel communication



NERSC franklin node
(and perhaps your laptop)
-4 cores per processor
-1 processor per node
-8 GB memory per node



OLCF jaguar node
-6 cores per processor
-2 processors per node
-16 GB memory per node

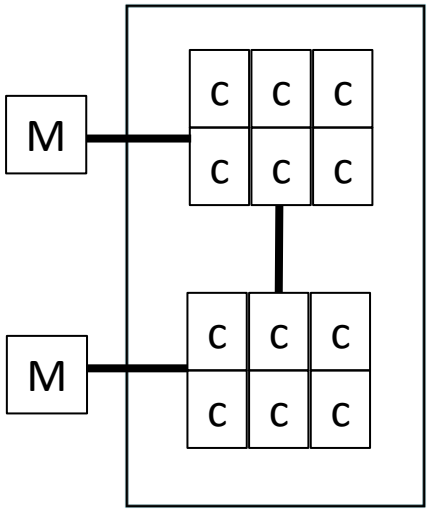


NERSC hopper node*
-12 cores per processor
-2 processors per node
-32 or 64 GB memory per node

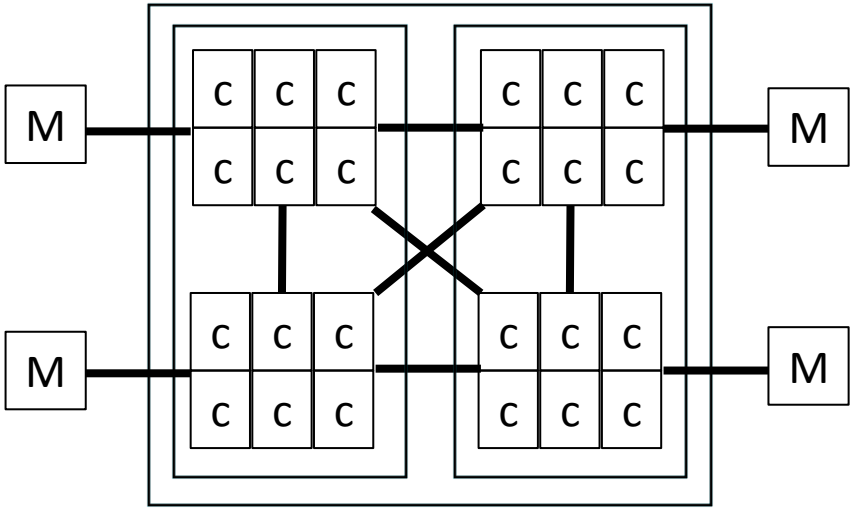
*On hopper, a NUMA (non-uniform memory access) node is 6 cores on one half of a processor

Beware Latency

- Note: For some architectures, due to latency, a core will access memory slower if not directly connected to it



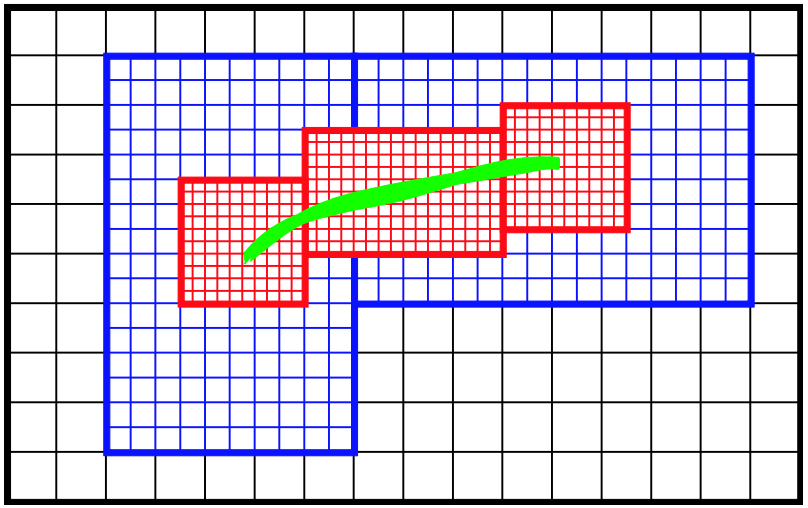
OLCF jaguar node
-6 cores per processor
-2 processors per node
-16 GB memory per node



NERSC hopper node
-12 cores per processor
-2 processors per node
-32 or 64 GB memory per node

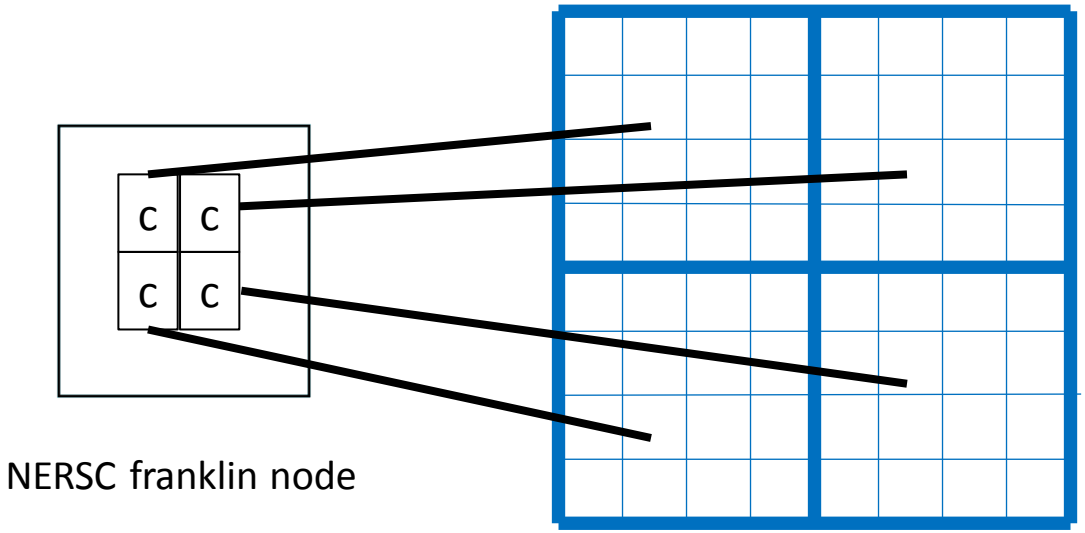
Finite Volume Framework with MPI

- Here we assume a finite-volume, block-structured approach to parallelism.
- Using pure MPI, we assign each block of data to a core.
  - Each block of data has ghost cells. We use MPI parallel communication to fill ghost cells.



Basic Idea of Threads

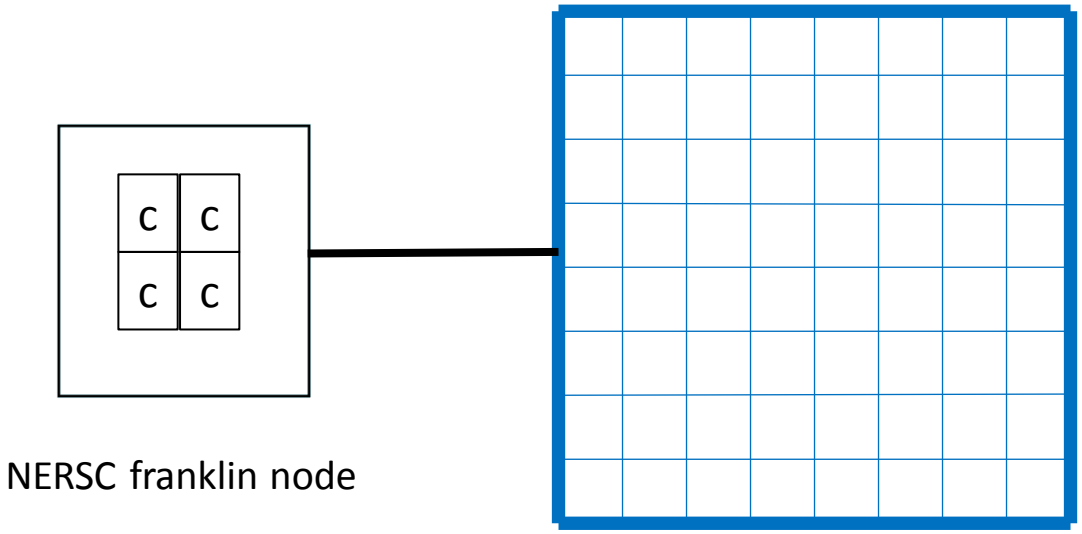
- Say you have four grids. In a pure MPI approach you would assign each grid to a core.



NERSC franklin node

Basic Idea of Threads

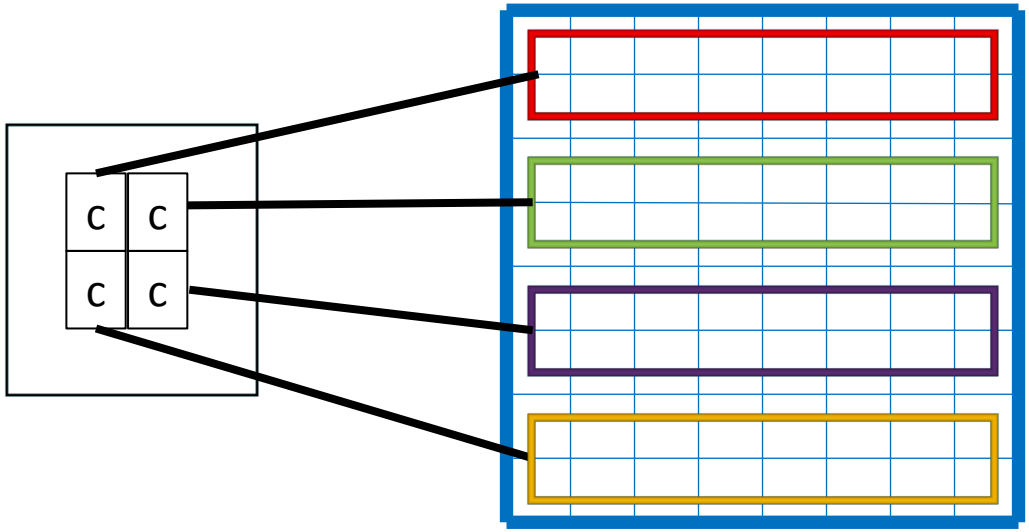
- Say you have four grids. In a pure MPI approach you would assign each grid to a core.
- Instead, assign a single, larger block of data to a node rather than a core …



NERSC franklin node

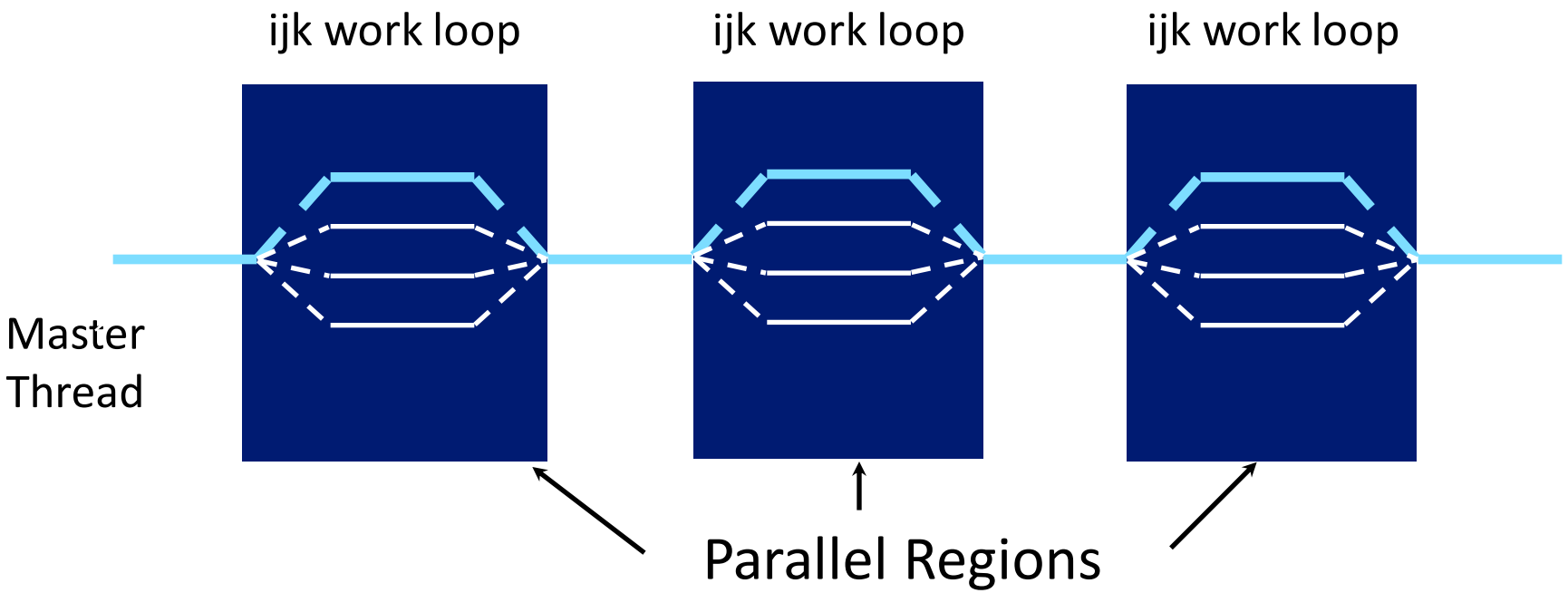
Basic Idea of Threads

- Say you have four grids. In a pure MPI approach you would assign each grid to a core.
- Instead, assign a single, larger block of data to a node rather than a core …
- … and then spawn a thread on each core to simultaneously work on the data



Basic Idea of Threads

- A master thread spawns a team of threads, then closes the threads when the loop is done

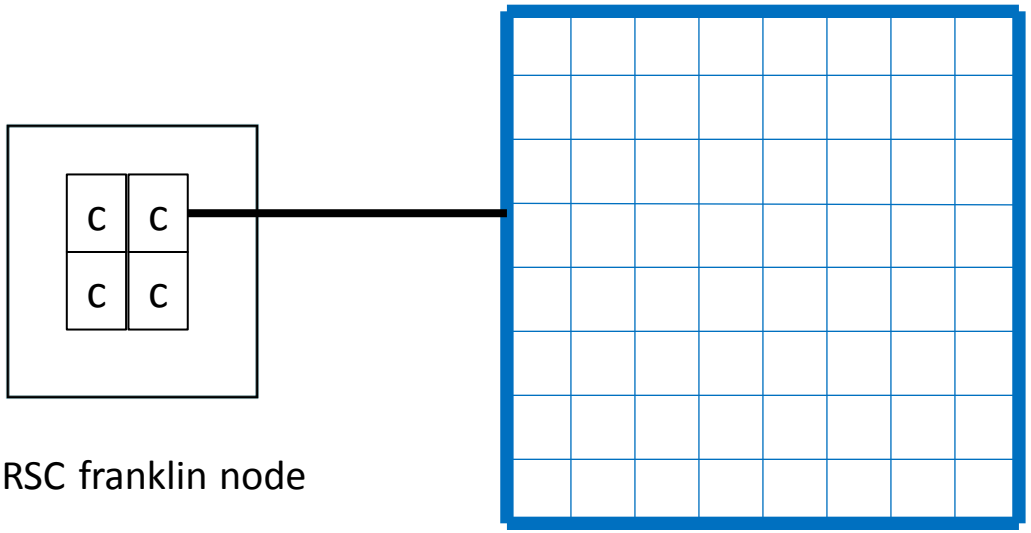


OpenMP Implementation

- We implement OpenMP at the loop level. Let's say we had a work loop (Fortran90)
  - A single core performs work on this grid.

```
! simple work loop

do j=1,8
   do i=1,8
      "do work"
   end do
end do
```

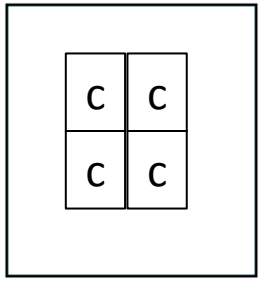


NERSC franklin node

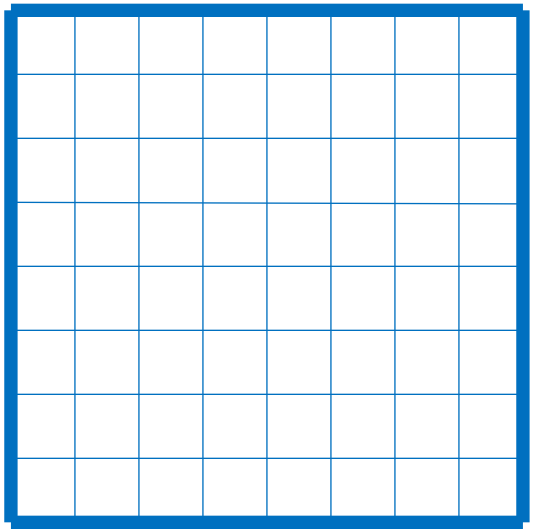
How do we do this?

- Add “omp directives”, which look like comments, but with the correct flags, the compiler will recognize them.
  - For PGI compilers (PrgEnv-pgi), we will compile with “ftn –mp=nonuma”, as documented on the NERSC web site:
    www.nersc.gov/users/computational-systems/hopper/programming/compiling-codes/

```
! simple work loop
!$omp parallel do private(j,i)
do j=1,8
   do i=1,8
      “do work”
   end do
end do
!$omp end parallel do
```



NERSC franklin node



OpenMP Implementation

- The OpenMP directives we add to our code basically tell each thread/core to do the following:

```
do j=1,2
   do i=1,8
      "do work"
   end do
end do
```

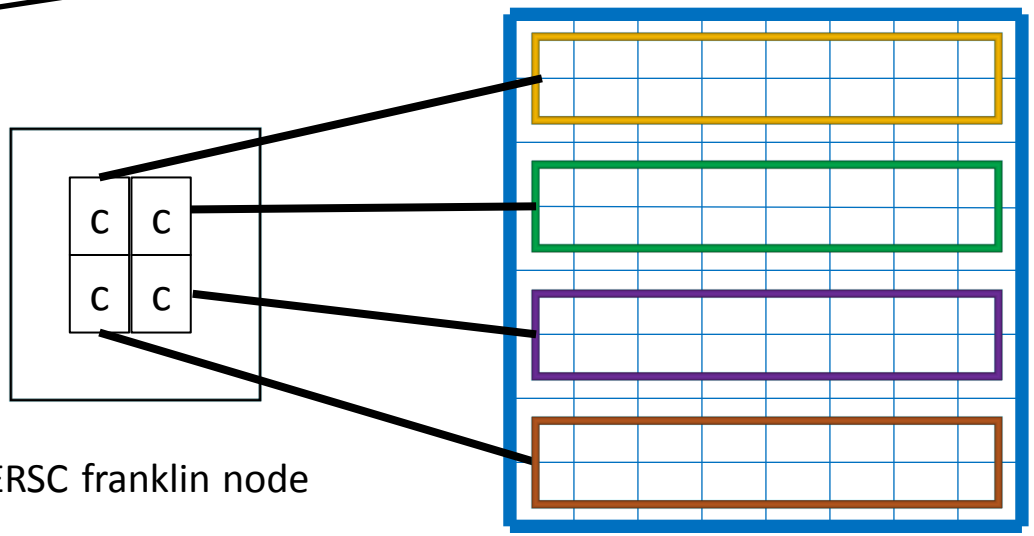
```
do j=3,4
   do i=1,8
      "do work"
   end do
end do
```

```
do j=5,6
   do i=1,8
      "do work"
   end do
end do
```

```
do j=7,8
   do i=1,8
      "do work"
   end do
end do
```

break up the first "do" loop encountered

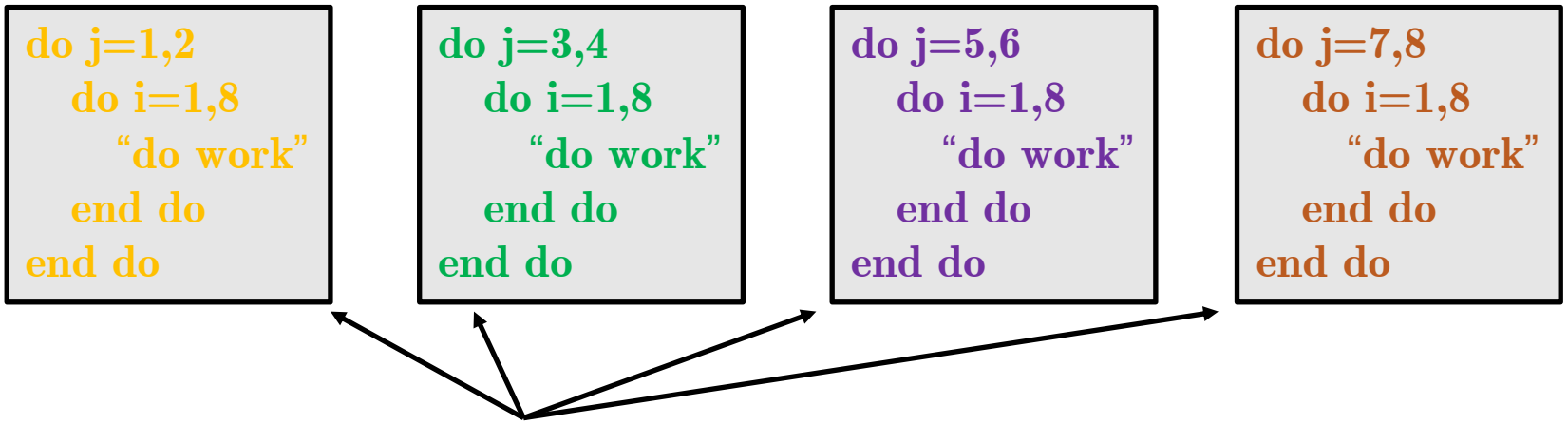
```
! simple work loop
!$omp parallel do private(j,i)
do j=1,8
   do i=1,8
      "do work"
   end do
end do
!$omp end parallel do
```



NERSC franklin node

OpenMP Implementation

- The OpenMP directives we add to our code basically tell each thread/core to do the following:




```
! simple work loop
!$omp parallel do private(j,i)
do j=0,7
   do i=0,7
      "do work"
   end do
end do
!$omp end parallel do
```

- "private" means each thread gets its own, uninitialized copy
  - "j" is optional, i.e., private by default
  - Order not important – I tend to list things in the order they appear

Lab Tutorials

- On hopper, copy the contents of
  /project/projectdirs/training/HIPACC_2011/nonaka/OpenMP_tutorial/
  to your home directory or scratch space on hopper

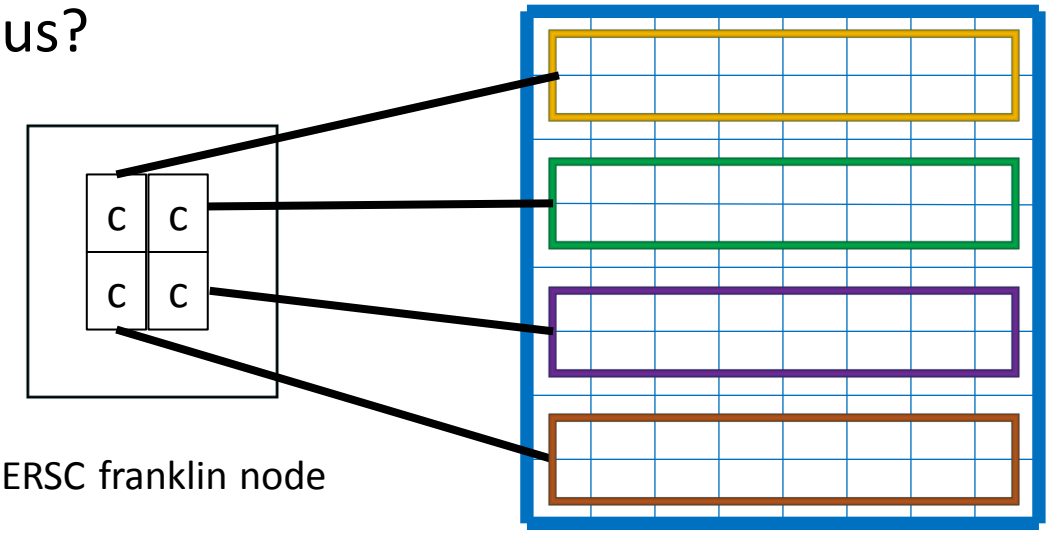
- Files:
  - OpenMP_tutorial_part1.f90
  - OpenMP_tutorial_part2.f90
  - OpenMP_tutorial_part3.f90
  - hopper.run
  - hopper_6threads.run
  - hopper_12threads.run
  - hopper_24threads.run
  - hopper_6threads_ann.run

OpenMP_tutorial_part1.f90

- Compile and run without threads
  - “ftn OpenMP_tutorial_part1.f90”
  - “qsub hopper.run”
- Add OpenMP directive to ijk loop
- Compile and run with threads
  - “ftn –mp=nonuma OpenMP_tutorial_part1.f90”
  - “qsub hopper_6threads.run”
  - “qsub hopper_12threads.run”
  - “qsub hopper_24threads.run”
- Compare solution and run time between all four runs.

Why Threads?

- In this previous cartoon, each core is still doing work on exactly 8 computational grid cells, so what's the advantage?
- Since memory is shared within a node, we can assign a large grid to a node, rather than more smaller grids to individual cores
  - Allows for fewer, larger grids.
    So how does this help us?



NERSC franklin node

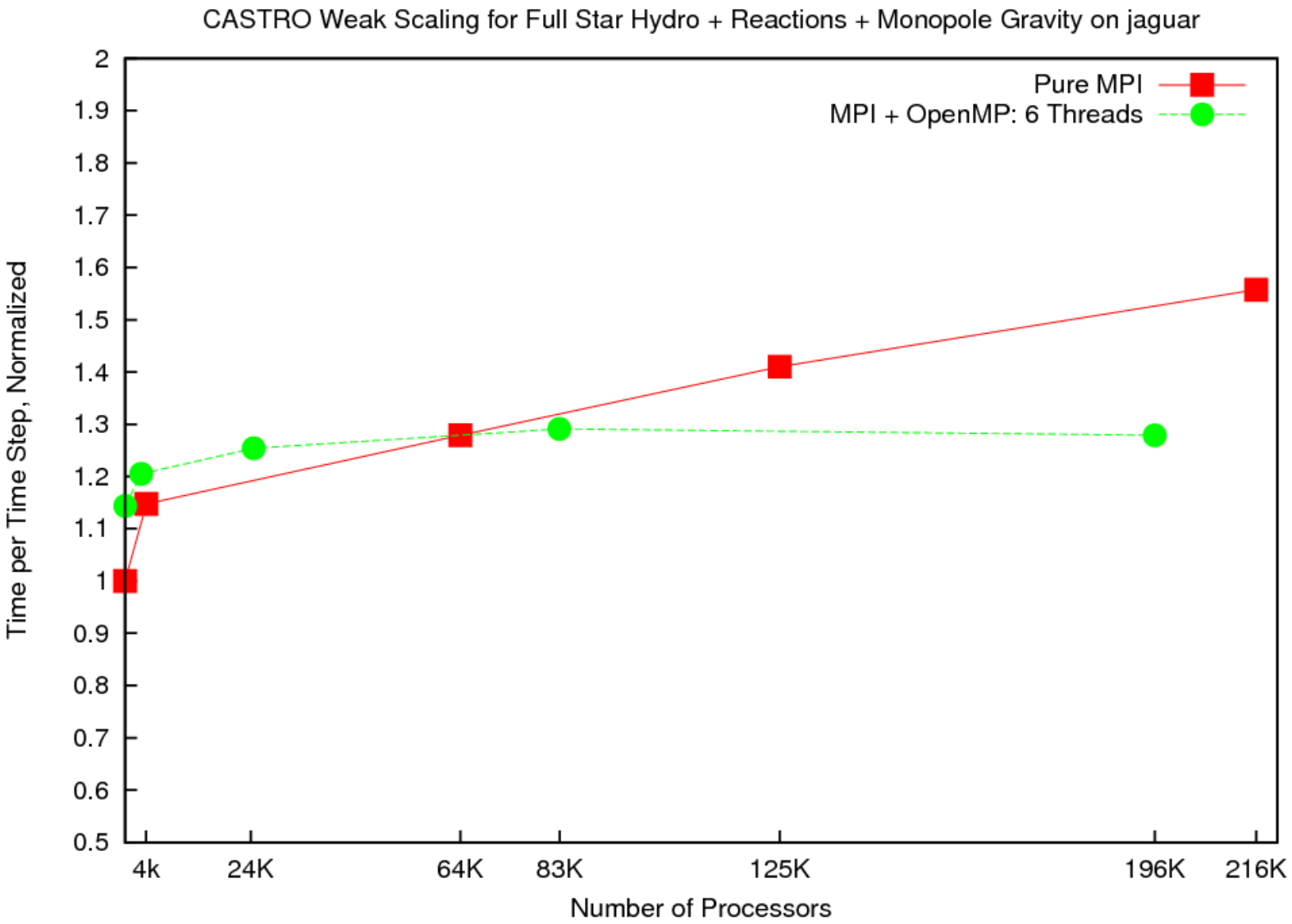
Why Fewer, Larger Grids?

- How does having fewer, larger grids help performance?
  - Fewer ghost cells, so less overall memory used in your program
  - Less memory overhead to keep track of grid structure and parallel communication patterns
  - Fewer MPI processes, which speeds up parallel communication
  - Communicating fewer, larger blocks of ghost cell data, which speeds up parallel communication
- Depending on your problem, some advantages are more important than others.
  - Complex reaction networks: memory limited
  - Linear Solvers: parallel communication limited

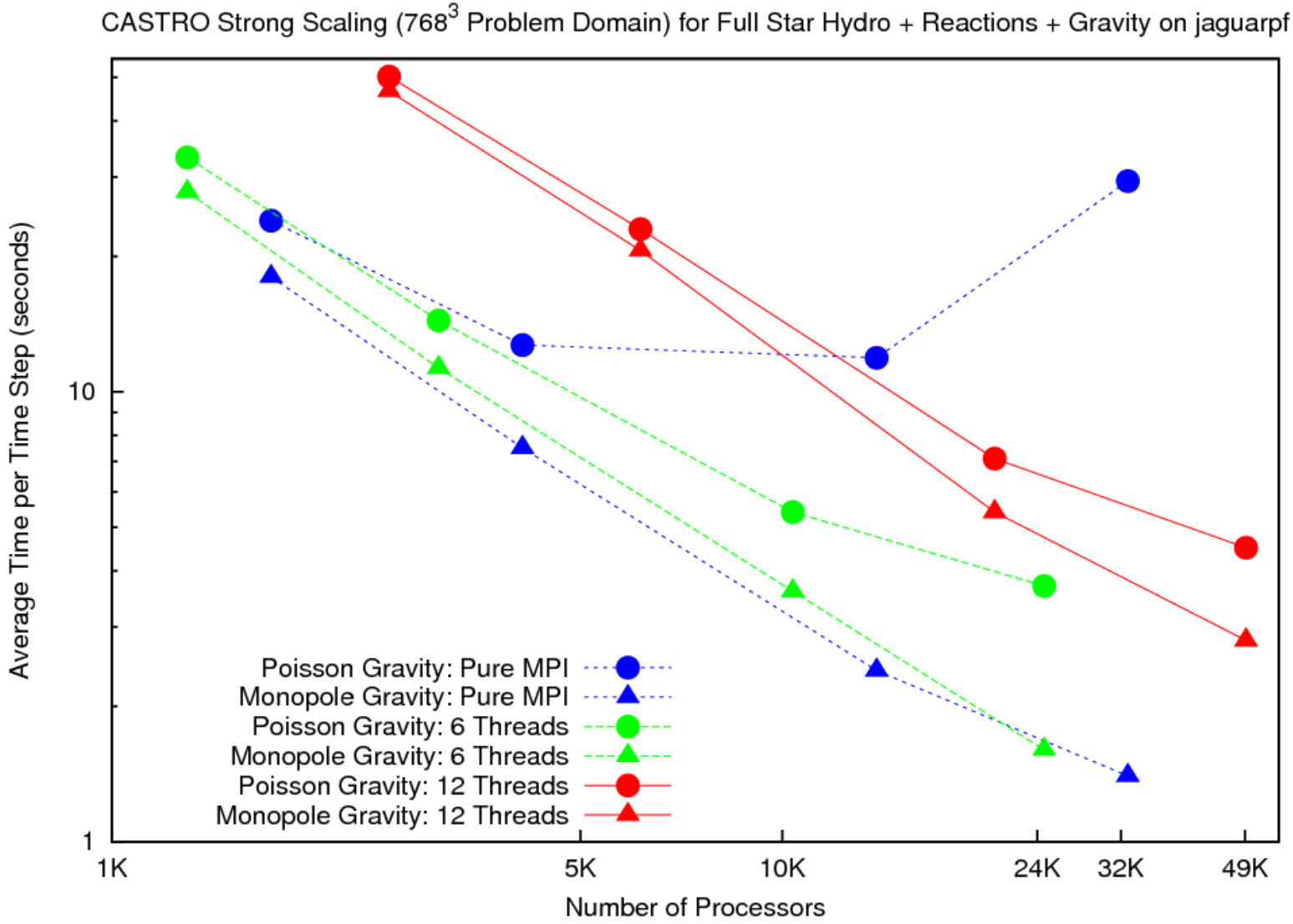
CASTRO and MAESTRO Success

- In CASTRO and MAESTRO, our bottleneck for performance has been parallel communication time, and not memory usage (so far), so we have focused more on using threads for scalability.

CASTRO Weak Scaling



CASTRO Strong Scaling



Great Things About OpenMP

- Since you can “thread” your code incrementally (i.e., add OpenMP directives to one loop at a time), you can test to make sure your results stay identical after “threading” any one loop.
- You can also measure the performance gains (speed of execution) due to threading that one loop.

Things to Keep in Mind

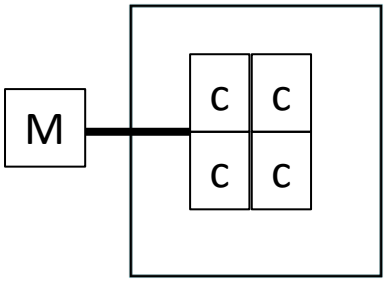
- It takes a finite amount of time for your code to spawn and close threads at runtime, so you will not always see a speed boost.
  - Example: Running CASTRO with only hydro and reactions for smallish problems (a few thousand cores) and 6 threads only speeds up by a factor of 3 as compared to pure MPI with $1/6^{th}$ the total number of cores.
  - You will (generally) be able to run faster given the same number of cores using threads if your problem uses linear solvers (Poisson gravity, Elliptic pressure/projection solves) and you have more than 1,000 MPI processes since communication is so time-intensive.
  - Sometimes you will have to run with threads and take a performance hit if memory is of concern.
  - Sometimes you will have to run with threads and take an efficiency hit if you just want the code to run faster.

Things to Keep in Mind

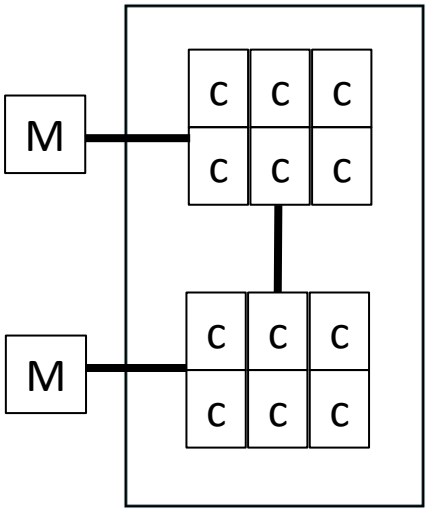
- If you are doing very little work within your i,j,k loop (like simply doing a copy or adding two numbers), threads can actually slow down your program since they take some time to spawn and close.
  - Also applies if your grid is small – there just isn't that much floating point work to do.

Things to Keep in Mind

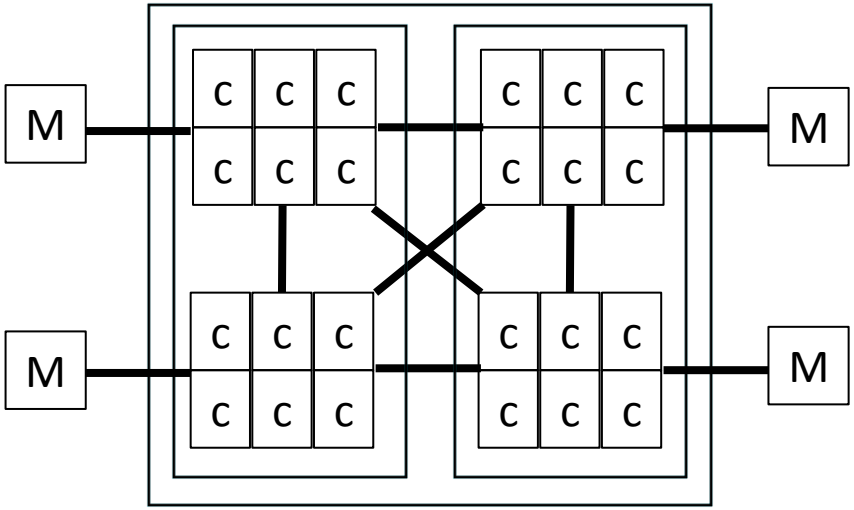
- The number of threads you use for optimal performance is architecture-dependent.
  - Rule of thumb: try not to make a thread reach for memory it's not directly connected to



NERSC franklin node
(and perhaps your laptop)
-4 cores per processor
-1 processor per node
-8 GB memory per node



OLCF jaguar node
-6 cores per processor
-2 processors per node
-16 GB memory per node



NERSC hopper node
-12 cores per processor
-2 processors per node
-32 or 64 GB memory per node

OpenMP_tutorial_part2.f90

- Compile and run without threads

- Add OpenMP directive to ijk loop
- Compile and run with 6, 12, and 24 threads
  - Compare run times between all four runs

- Add more work to the loop (uncomment the line I put in there)
- Compile and run with 6, 12, and 24 threads
  - Compare run times between these three runs

- Add even more work to the loop (uncomment the line I put in there)
- Compile and run with 6, 12, and 24 threads
  - Compare run times between these three runs

Shared vs. Private

- In an OpenMP do loop, all variables are considered "shared" unless you explicitly label them as "private".
  - Shared means all threads access the same copy of the variable owned by the master thread
  - Private means each thread gets its own uninitialized copy
- In this example both "y" and "z" are private, whereas "x" is shared.

```
integer i,j
double precision a(1:8,1:8)
double precision x,y,z

x = 10.0d0

! initialize "a"
!$omp parallel do private(j,i,y,z)
do j=1,8
   do i=1,8

      y = x*(i+j)
      z = i*j
      a(i,j) = y + z

   end do
end do
!$omp end parallel do
```

Reductions: Sum, Min, and Max

- Special syntax for reductions (sum, minimum, maximum)
- Need to give sum, minval, and maxval initial values or else it will begin with the default values of
  - sum: 0.0
  - minval: 0.0
  - maxval: 1.0

```
integer i,j
double precision a(1:8,1:8)
double precision sum, minval, maxval

minval = HUGE
maxval = -HUGE

! assume "a" has been initialized
!$omp parallel do private(j,i) reduction(+:sum) &
!$omp reduction(min:minval) reduction(max:maxval)
do j=1,8
   do i=1,8

      sum = sum + a(i,j)
      minval = min(minval,a(i,j))
      maxval = max(maxval,a(i,j))

   end do
end do
!$omp end parallel do
```

OpenMP_tutorial_part3.f90

- Compile and run without threads

- Add OpenMP directive to each ijk loop
- Compile and run with 6, 12, and 24 threads
  - Compare solution and run times between all four runs

- See what happens if you “mess up”. Try running the following with threads, and see what happens to the answer.
  - In the second ijk loop, don’t make var1 and/or var2 private
  - In the third ijk loop, make x private
  - In the fourth ijk loop, either make sum, minval, and/or maxval private, or don’t include them in the OpenMP directive at all

Remarks

- These tutorials cover everything you need to know about threading the hydro codes you wrote this week.

- These tutorial cover 99% of what we needed to know to put OpenMP into CASTRO and MAESTRO.
  - Time permitting, I'll show you some examples of the two other OpenMP directives we used
    - The IF clause
    - THREADPRIVATE for common blocks in EOS and reaction networks
  - Time permitting, I'll introduce some other OpenMP directives including
    - FIRSTPRIVATE, LASTPRIVATE, SINGLE

Projects

- Thread the advance_2d.f90 subroutine from Ann's hydro code example /project/projectdirs/training/HIPACC_2011/almgren/BoxLibTest
  - To build, "module swap PrgEnv-pgi PrvEnv-gnu", then "make USE_OMP=TRUE". You will get an executable with a new name.
  - Run with 6 threads per MPI process (i.e., 6 times as many cores) and see how performance compares to pure MPI
  - Using some of the principles we discussed today, there are several optimizations within advance_2d.f90 that can make the threaded version of the code run about 7-8% faster. See if you can find them.
- Try threading some routines in your hydro code to compare performance (make sure to check the answer remains the same)!