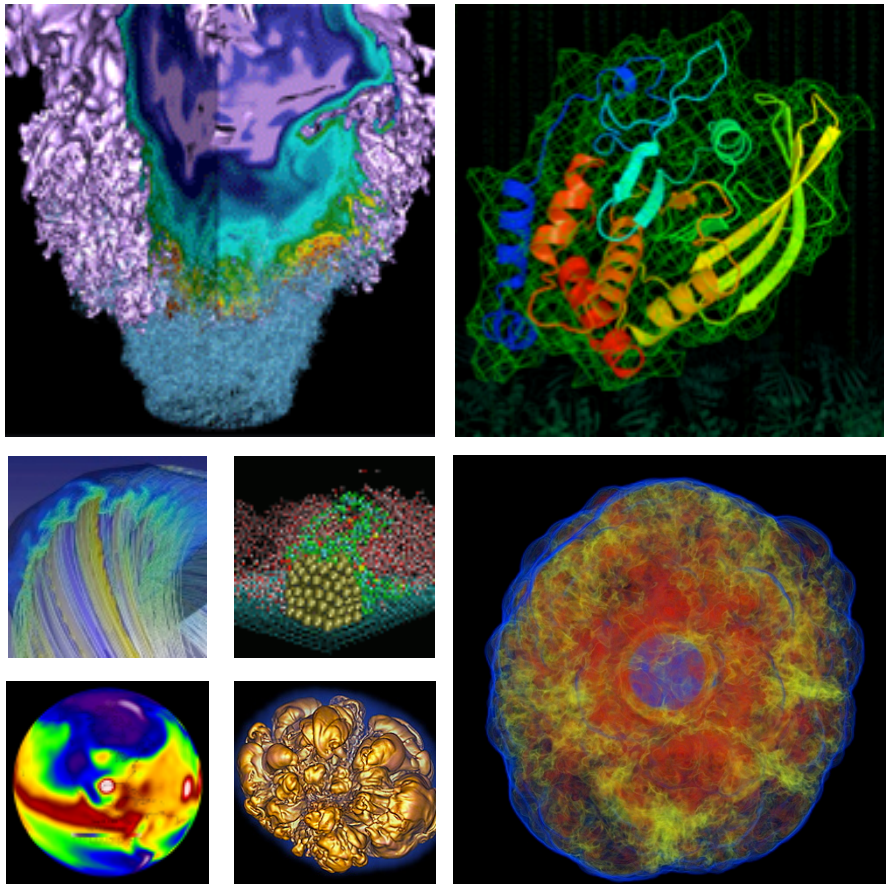
Preparing Codes (and Nyx) for Intel Xeon Phi on Cori



Brian Friesen

2017 17 Apr

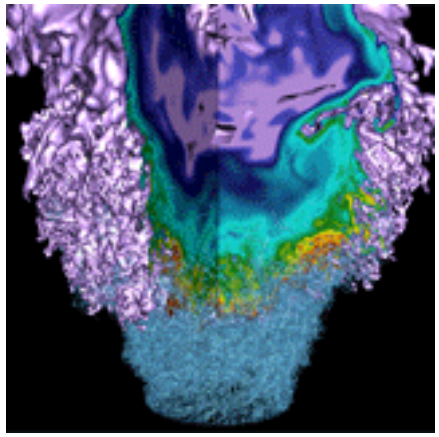
Cori is the newest system at NERSC

- **Cray XC40**
- **"Partitioned" system**
  - 2004 nodes of Intel Xeon E5-2698 v3 ("Haswell")
  - 9688 nodes of Intel Xeon Phi 7250 ("Knights Landing")
- **Xeon nodes – 2 sockets, 16 cores/socket**
  - "Big cores" – 2.3 GHz, L1/L2/L3 cache (40 MB)
  - 128 GB DDR4 DRAM
- **Xeon Phi nodes – 1 socket, 68 cores/socket**
  - "Little cores" – 1.2 GHz, only L1/L2 cache (no L3)
  - 96 GB DDR4 DRAM + 16 GB high-bandwidth, multi-channel DRAM ("MCDRAM")
- **Cori's successor is coming in 2020, so stay tuned** …

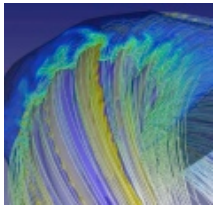
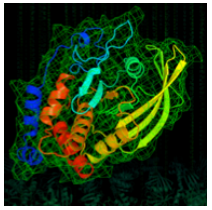
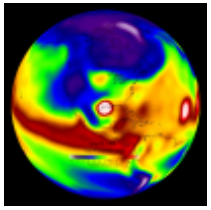
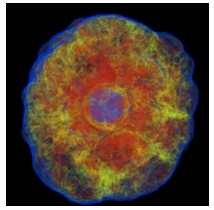
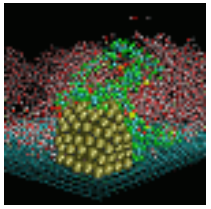
Upshot: Computers are hard

- **Xeon Phi represents a smooth(ish) transition from the era of a few big, fast CPUs to lots of small, lightweight CPUs**
- **But getting codes to run well on Xeon Phi is hard**
  - Vectorization is no longer an option for good performance – it is required
  - More necessary than before to use programming models which exploit on-node parallelism (as opposed to bulk-synchronous parallelism viz. MPI)
    - Lots of these – OpenMP, OpenACC, Cilk, Threaded Building Blocks, Kokkos, RAJA, HPX, UPC, etc.

Preparing codes for Xeon Phi



NERSC

Preparing for Xeon Phi at NERSC

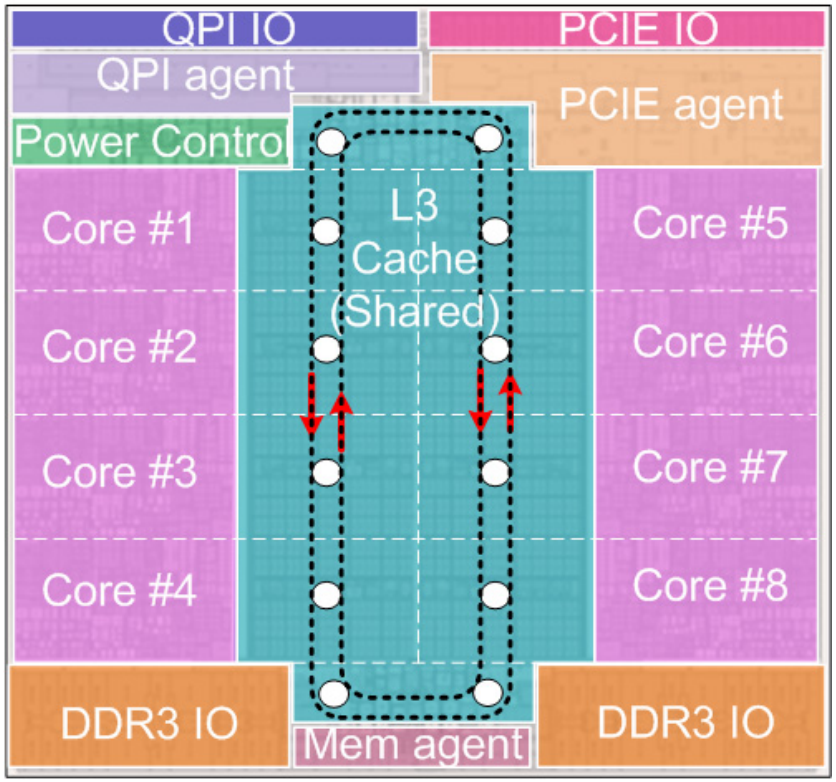
- **NERSC started "preparing" for Cori in 2014**
  - NERSC Exascale Science Application (Readiness) Program
  - Early access to pre-production hardware
  - *LOTS* of training with engineers from Intel and Cray
  - Many 3-day on-site "dungeon sessions" at Intel site in Portland, OR
- **Most developers use Edison as a starting point**
  - 2 sockets, 2 cores/socket
  - Intel Xeon E5-2695 v2 ("Ivy Bridge") @ 2.4 GHz
  - L1/L2/L3 caches (30 MB)

How does one "prepare" for Cori?

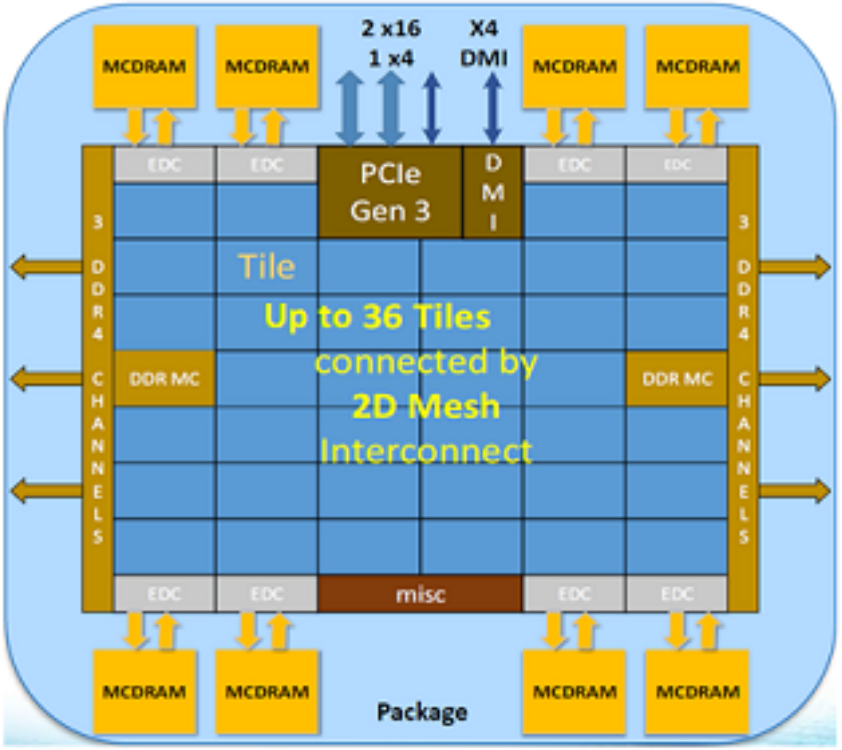
- **If you plan on using Xeon (Haswell): pretty much nothing**
  - Moore's law is not dead yet
  - Core count/node vs Edison has increased (24 -> 32)
  - Clock speeds basically the same (2.4 GHz -> 2.3 GHz)
  - Cache size has increased (30 MB -> 40 MB)
  - SIMD units have grown (128-bit -> 256-bit)
  - DRAM/node has increased (64 GB -> 128 GB)
  - "Recompile and run"
- **OTOH, If you plan on using Xeon Phi (KNL): you have some work to do**

Preparing for Cori

Sandy Bridge Overview



Knights Landing Overview



Xeon (Edison) vs Xeon Phi (Cori)

- **Core counts/node have increased (24 -> 68)**
- **Clock speeds have decreased (2.4 GHz -> 1.2 GHz)**
- **Cache/node is about the same, but cache/core has decreased (1.3 MB -> 512 KB)**
- **SIMD units have grown (128-bit -> 512-bit)**
- **DRAM/node has increased, but DRAM/core has decreased**
  - 2.7 GB -> 1.6 GB, and only **235 MB** if you want just the high-bandwidth memory
- **Without significant optimization effort, many codes will run slower on KNL than on HSW**
  - But with optimization, many codes can run faster!

What does this mean for developers?

- **Nodes are getting "more parallel" – more cores, wider vectors**
- **A code that runs fast on Cori needs to express lots of parallelism, everywhere, all the time**
- **You have to parallelize *everything*, or Amdahl's law will bite you harder than it used to**
  - Much bigger effect on GPUs than on Xeon Phi
- **The hierarchy of parallelism is deepening**
  - In the old days, MPI was "enough"
  - In general, not anymore – you need to express **on-node** parallelism too, e.g., threading, vectorization

Vectorizability

- **Each Xeon/Xeon Phi CPU has at least one "vector processing unit" (VPU) that can process several types of instructions at once**
  - +-*/, exp, log, etc.
- **The VPU "width" tells you how many instructions it can process at once**
  - On Xeon Phi, each VPU is 512 bits -> 8 double precision operations per instruction (64 bits each)

Vectorizability

- **Many types of operations can be vectorized:**
  - Dot product:

```
do i = 1, N
    a = a + b(i) * c(i)
end do
```

  - If b and c are double precision, then Xeon Phi can process this loop in just N/8 instructions
- **But some cannot:**
  - Ray tracing:

```
do i = 2, N
    a(i) = a(i-1) * exp(-b(i)/c(i))
end do
```

A happy compiler report

```
120      1.            program main
121      2.              use, intrinsic :: iso_c_binding
122      3.              implicit none
123      4.
124      5.              integer(C_INT), parameter :: N = 1024
125      6.              real(C_DOUBLE) :: a, b(N), c(N)
126      7.              integer(C_INT) :: i
127      8.
128      9.              a = 0.0
129     10.    Vpr2--<   do i = 1, N
130     11.    Vpr2        b(i) = real(i*2, C_DOUBLE)
131     12.    Vpr2        c(i) = real(i*3, C_DOUBLE)
132     13.    Vpr2-->   end do
133     14.
134     15.              write(*, '(2es13.3e2)') b(1), c(1)
135     16.
136     17.    Vr4---<   do i = 1, N
137     18.    Vr4         a = a + b(i) * c(i)
138     19.    Vr4--->   end do
139     20.
140     21.              write (*, '(a,es13.3e2)') 'result: ', a
141     22.
142     23.            end program main
143
144 ftn-6005 ftn: SCALAR MAIN, File = test_dot_product.f90, Line = 10
145   A loop starting at line 10 was unrolled 2 times.
146
147 ftn-6209 ftn: VECTOR MAIN, File = test_dot_product.f90, Line = 10
148   A loop starting at line 10 was partially vectorized.
149
150 ftn-6005 ftn: SCALAR MAIN, File = test_dot_product.f90, Line = 17
151   A loop starting at line 17 was unrolled 4 times.
152
153 ftn-6204 ftn: VECTOR MAIN, File = test_dot_product.f90, Line = 17
154   A loop starting at line 17 was vectorized.
155
```

A sad compiler report

```
119
120      1.           program main
121      2.             use, intrinsic :: iso_c_binding
122      3.             implicit none
123      4.
124      5.             integer(C_INT), parameter :: N = 1024
125      6.             real(C_DOUBLE) :: b(N), c(N)
126      7.             integer(C_INT) :: i
127      8.
128      9.     A-->    c(:) = real(i*3, C_DOUBLE)
129     10.
130     11.             b(1) = 0.0
131     12.
132     13.   + r2--<   do i = 2, N
133     14.     r2        b(i) = b(i-1) * exp(-c(i)*b(i-1))
134     15.     r2-->   end do
135     16.
136     17.             write (*, '(a,es13.3e2)') 'result: ', b(N)
137     18.
138     19.           end program main
139
140 ftn-6202 ftn: VECTOR MAIN, File = test_dot_product_sad.f90, Line = 9
141   A loop starting at line 9 was replaced by a library call.
142
143 ftn-6005 ftn: SCALAR MAIN, File = test_dot_product_sad.f90, Line = 13
144   A loop starting at line 13 was unrolled 2 times.
145
146 ftn-6254 ftn: VECTOR MAIN, File = test_dot_product_sad.f90, Line = 13
147   A loop starting at line 13 was not vectorized because a recurrence was found on "b" at line 14.
148
```

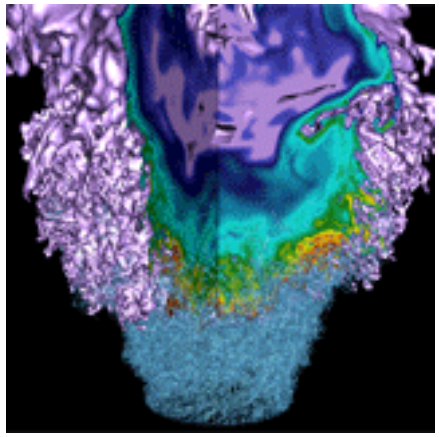
Why vectorizing is harder than threading

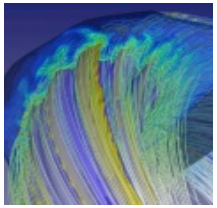
- **Threads within a parallel region can operate entirely independently**
  - Threads can execute this loop in parallel:

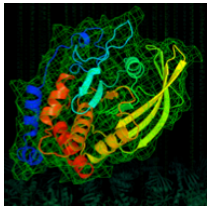
```
do i = 1, N
    if (i < N/2) call this_func(i);
end do
```

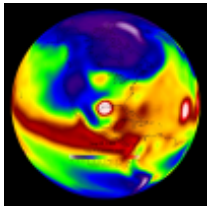
- **Vector instructions must operate in lock-step**
  - Different "lanes" within a vector instruction cannot have execute different logic (with a few exceptions)

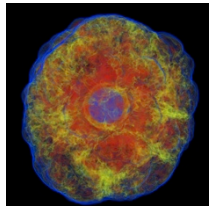
Preparing Nyx for Xeon Phi

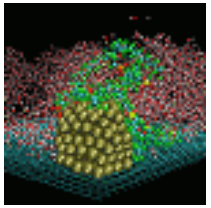






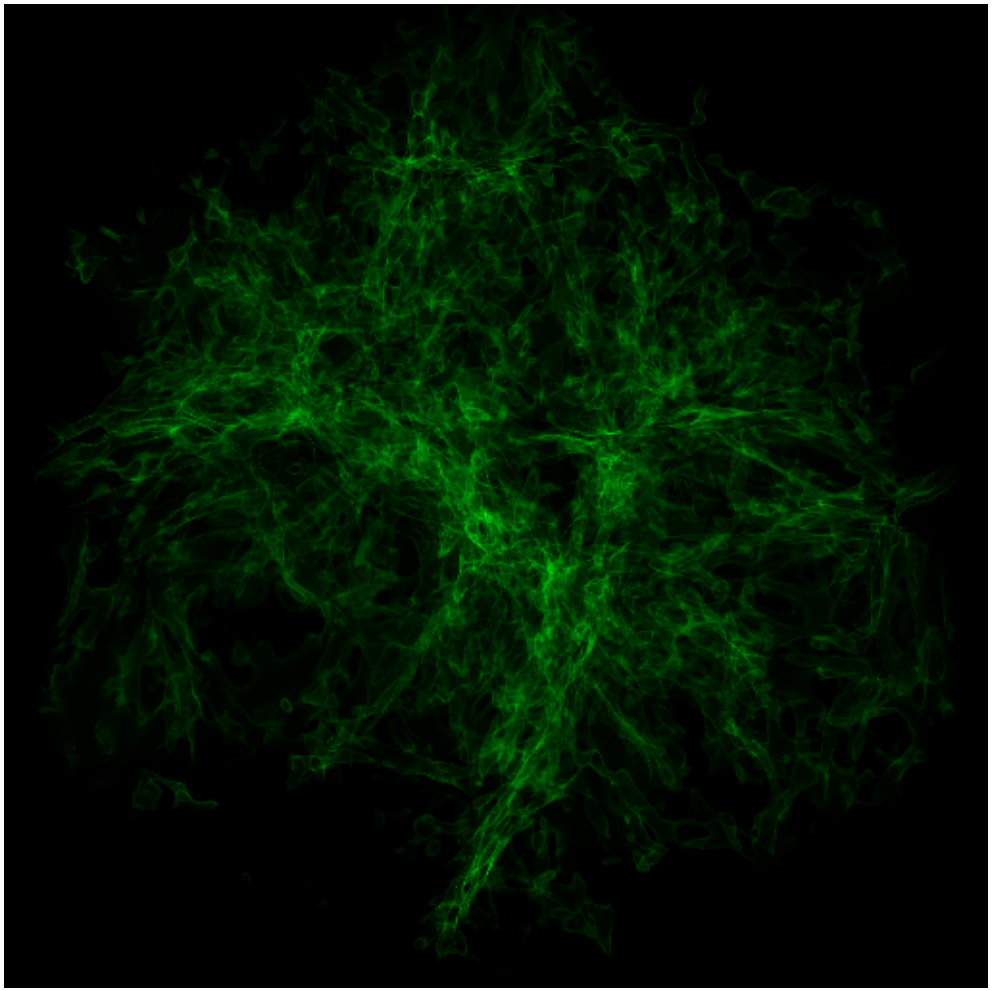






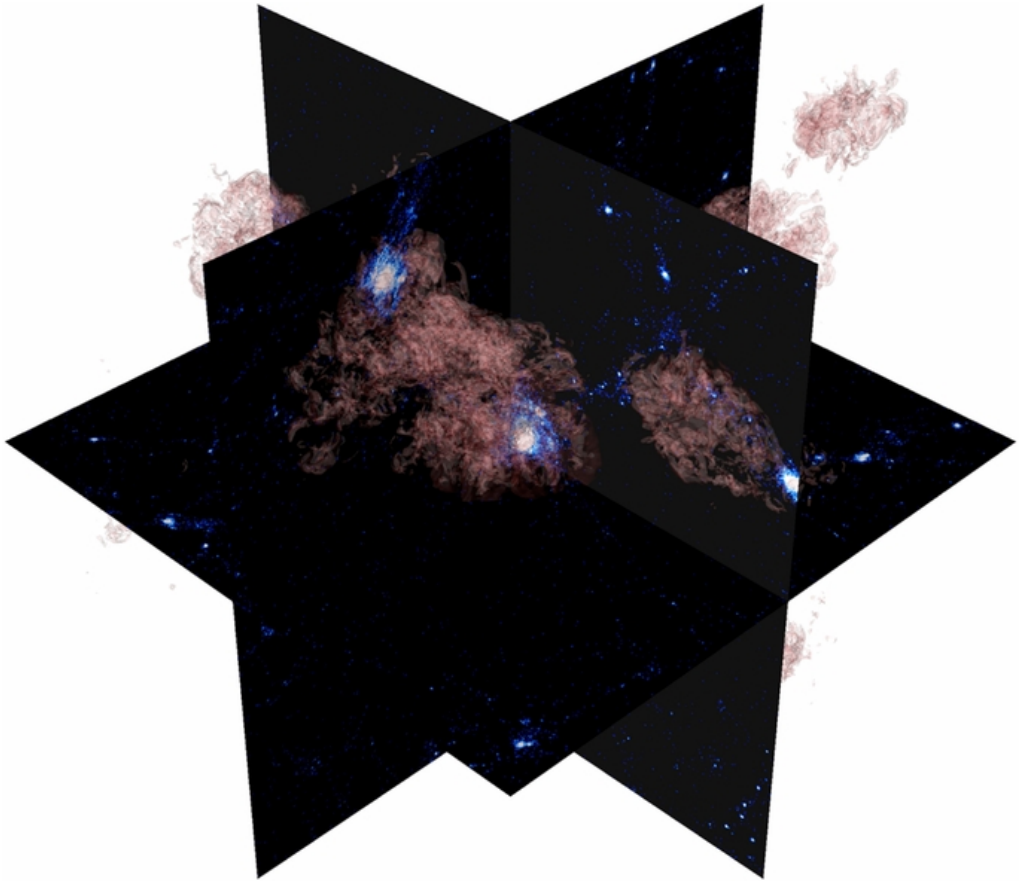
Nyx Overview

- **3-D hybrid *N*-body/ compressible gasdynamics code for cosmological simulations**
  - Collisionless dark matter, radiation, self-gravity
- **Supports adaptive meshes with subcycling**
- **Based on AMReX framework (formerly BoxLib)**



Nyx algorithm components

- **Compressible gas advance***
  - 2nd order unsplit PPM
- **RT (local H/C)***
  - Implicit time integration of stiff ODE per cell
- **Self-gravity**
  - Geometric multigrid
- **Dark matter particle-mesh deposition/ interpolation**
  - Cloud-in-cell method



Challenges for Nyx on Xeon Phi

- **Primarily the lack of *vectorization***
  - If a code does not vectorize, it can *never* run faster on KNL than on Haswell, no matter how optimized it is otherwise
- **Two major serial bottlenecks in Nyx:**
  - Compressible hydro
    - Mostly point-wise operations
    - Complex loops that compilers can't vectorize
  - Local heating/cooling
    - Implicit integration of stiff ODEs (one per cell)
    - Integrator is implicit and adaptive − vectorizing integration across cells is non-trivial (but not impossible)

Original compressible flow code

```
do i, j, k:
    compute(i,j,k);      // compute lots of temporary
    many(i,j,k);         // values in order to compute
    things(i,j,k);       // one value of something
    if (something):      // per cell (e.g., pressure)
        do_this(i,j,k);
    else
        do_that(i,j,k);
    . . .
```

1. ssh cori (ssh)

IPython: Users/friese... ⌘1 | ..uired_reading (zsh) ⌘2 | ~ (zsh) ⌘3 | ~ (zsh) ⌘4 | cori (ssh) ⌘5

```
421
422           ! compute on slab above
423           k = k3d+1
424           dsc = HALF * (s(i,j,k+1) - s(i,j,k-1))
425           dsl = TWO  * (s(i,j,k  ) - s(i,j,k-1))
426           dsr = TWO  * (s(i,j,k+1) - s(i,j,k  ))
427           if (dsl*dsr .gt. ZERO) &
428                dsvlp(i,j) = sign(ONE,dsc)*min(abs(dsc),abs(dsl),abs(dsr))
429
430           ! compute on current slab
431           k = k3d
432           dsc = HALF * (s(i,j,k+1) - s(i,j,k-1))
433           dsl = TWO  * (s(i,j,k  ) - s(i,j,k-1))
434           dsr = TWO  * (s(i,j,k+1) - s(i,j,k  ))
435           if (dsl*dsr .gt. ZERO) &
436                dsvl(i,j) = sign(ONE,dsc)*min(abs(dsc),abs(dsl),abs(dsr))
437
438           ! interpolate to lo face
439           k = k3d
440           sm(i,j) = HALF*(s(i,j,k)+s(i,j,k-1)) - SIXTH*(dsvl(i,j)-dsvlm(i,j))
441           ! make sure sedge lies in between adjacent cell-centered values
442           sm(i,j) = max(sm(i,j),min(s(i,j,k),s(i,j,k-1)))
443           sm(i,j) = min(sm(i,j),max(s(i,j,k),s(i,j,k-1)))
444
445           ! interpolate to hi face
446           k = k3d+1
447           sp(i,j) = HALF*(s(i,j,k)+s(i,j,k-1)) - SIXTH*(dsvlp(i,j)-dsvl(i,j))
448           ! make sure sedge lies in between adjacent cell-centered values
449           sp(i,j) = max(sp(i,j),min(s(i,j,k),s(i,j,k-1)))
450           sp(i,j) = min(sp(i,j),max(s(i,j,k),s(i,j,k-1)))
451
452           if (ppm_flatten_before_integrals == 1) then
453              ! flatten the parabola BEFORE doing the other
454              ! monotonization -- this is the method that Flash does
455              sm(i,j) = flatn(i,j,k3d)*sm(i,j) + (ONE-flatn(i,j,k3d))*s(i,j,k3d)
456              sp(i,j) = flatn(i,j,k3d)*sp(i,j) + (ONE-flatn(i,j,k3d))*s(i,j,k3d)
457           endif
458
459           ! modify using quadratic limiters
```

459,11 37%

What's wrong with this?

- **From a human perspective, it's fine**
  - It's clean, readable, somewhat self-documenting
- **From a compiler perspective, it's not fine**
  - Compilers try to vectorize code automatically if they can figure out how
  - They try to evaluate every possible path for each datum in the loop in order to determine whether or not it can vectorize *and still produce the same result*
  - Complexity of this task grows exponentially with loop size, and most compilers force themselves to "give up" if they can't figure it out in a fixed amount of time
- **Compilers must always produce the correct result − that is not negotiable**
  - They will always err on the side of caution if they can't figure out whether a loop is safe to vectorize
  - they will not vectorize it unless they can ***prove*** that it will still give the same answer

How can we vectorize this? Several options

- **If you know the loop is vectorizable, use OpenMP to decorate it with a "#pragma omp simd" directive which tells compiler to vectorize the loop without checking whether it is safe**
  - Only works if you compile with OpenMP enabled, otherwise compiler will ignore it
  - Relies on you the developer ensuring that the vectorized loop still gives the right answer
- **Fission giant loops into a series of smaller ones**
  - Compilers are much better at evaluating vectorizability of small loops
  - This usually requires using more memory because one generally saves some temporary values between each series of loops

How can we vectorize this? Several options

- **If you're using Fortran (or C/C++ with Cilk), use array notation**
  - Instead of this:

```
do i = 1, n
    a(i) = b(i) * c(i)
end do
```

  - Do this:

```
a(1:n) = b(1:n) * c(1:n)
```

  - Or, if you set up your function arguments appropriately:

```
a = b*c
```

What did I do in Nyx?

- **Mostly switched to array syntax**
  - I don't like to rely on #pragma omp simd because it requires you to compile with OpenMP to vectorize the loop, even if you don't want to use OpenMP for threading
    - OTOH, using means you don't have to spend hours refactoring your loops to make them easy for the compiler to vectorize
  - What if we decide that OpenMP is not the best threading model, and switch to something else in the future? Then this loop won't vectorize anymore
  - My MO: *do it in the language if you can*
    - Programming models come and go, but languages tend to stick around

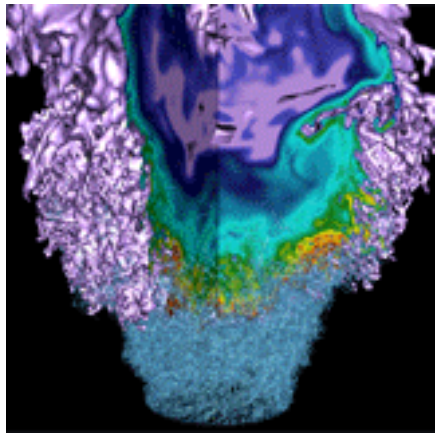
1. ssh cori (ssh)

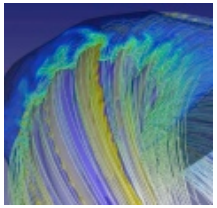
IPython: Users/friese... ⌘1 | ..uired_reading (z... ⌘2 | ~ (zsh) ⌘3 | ~ (zsh) ⌘4 | cori (ssh) ⌘5

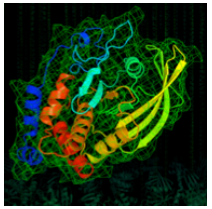
```
478         dsl = TWO  * (s(ilo1-1:ihi1+1,j,k  ) - s(ilo1-1:ihi1+1,j,k-1))
479         dsr = TWO  * (s(ilo1-1:ihi1+1,j,k+1) - s(ilo1-1:ihi1+1,j,k  ))
480         do i = ilo1-1, ihi1+1
481            if (dsl(i)*dsr(i) .gt. ZERO) &
482                 dsvlp(i,j) = sign(ONE,dsc(i))*min(abs(dsc(i)),abs(dsl(i)),abs(dsr(i)))
483         end do
484
485         ! compute on current slab
486         k = k3d
487         dsc = HALF * (s(ilo1-1:ihi1+1,j,k+1) - s(ilo1-1:ihi1+1,j,k-1))
488         dsl = TWO  * (s(ilo1-1:ihi1+1,j,k  ) - s(ilo1-1:ihi1+1,j,k-1))
489         dsr = TWO  * (s(ilo1-1:ihi1+1,j,k+1) - s(ilo1-1:ihi1+1,j,k  ))
490         do i = ilo1-1, ihi1+1
491            if (dsl(i)*dsr(i) .gt. ZERO) &
492                 dsvl(i,j) = sign(ONE,dsc(i))*min(abs(dsc(i)),abs(dsl(i)),abs(dsr(i)))
493         end do
494
495         ! interpolate to lo face
496         k = k3d
497         sm = HALF*(s(ilo1-1:ihi1+1,j,k)+s(ilo1-1:ihi1+1,j,k-1)) - SIXTH*(dsvl(ilo1-1:ihi1+1,j)-dsvlm(ilo1-1:ihi1+1,j))
498         ! make sure sedge lies in between adjacent cell-centered values
499         sm = max(sm,min(s(ilo1-1:ihi1+1,j,k),s(ilo1-1:ihi1+1,j,k-1)))
500         sm = min(sm,max(s(ilo1-1:ihi1+1,j,k),s(ilo1-1:ihi1+1,j,k-1)))
501
502         ! interpolate to hi face
503         k = k3d+1
504         sp = HALF*(s(ilo1-1:ihi1+1,j,k)+s(ilo1-1:ihi1+1,j,k-1)) - SIXTH*(dsvlp(ilo1-1:ihi1+1,j)-dsvl(ilo1-1:ihi1+1,j))
505         ! make sure sedge lies in between adjacent cell-centered values
506         sp = max(sp,min(s(ilo1-1:ihi1+1,j,k),s(ilo1-1:ihi1+1,j,k-1)))
507         sp = min(sp,max(s(ilo1-1:ihi1+1,j,k),s(ilo1-1:ihi1+1,j,k-1)))
508
509         if (ppm_flatten_before_integrals == 1) then
510            ! flatten the parabola BEFORE doing the other
511            ! monotonization -- this is the method that Flash does
512            sm = flatn(ilo1-1:ihi1+1,j,k3d)*sm + (ONE-flatn(ilo1-1:ihi1+1,j,k3d))*s(ilo1-1:ihi1+1,j,k3d)
513            sp = flatn(ilo1-1:ihi1+1,j,k3d)*sp + (ONE-flatn(ilo1-1:ihi1+1,j,k3d))*s(ilo1-1:ihi1+1,j,k3d)
514         endif
515
516         ! modify using quadratic limiters
```

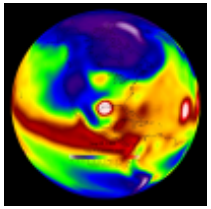
514,5 39%

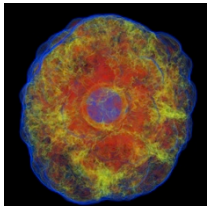
(backup)

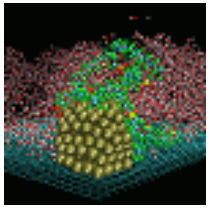












Branching in vector instructions (1/2)

- **Modern vector instructions (e.g., AVX-512 in Xeon Phi) do support limited branching among different lanes in a single vector instruction, e.g.,**

```
do i = 1, N
    if (i < N/2) then
        a(i) = i
    else
        a(i) = 2*i
end do
```

Branching in vector instructions (2/2)

- The VPU will actually evaluate *both* branches of the if-else clause, and keep only the correct branch once it actually computes the value of i
- Even though branching is possible, the efficiency of the VPU decreases as the branches get more complex
- Even modern VPUs don't support function calls in branches (unless they are inlined or explicitly declared SIMD-friendly)

National Energy Research Scientific Computing Center