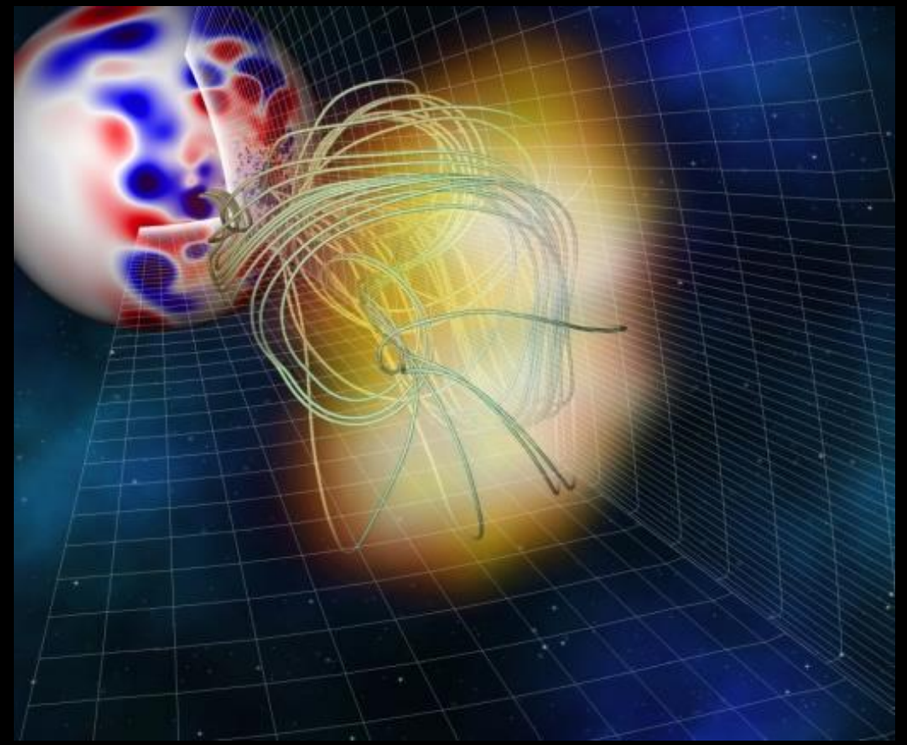
Potential Field Solutions of the Solar Corona: Converting a PCG Solver from MPI to MPI+OpenACC

Ronald M. Caplan,
Jon A. Linker, and Zoran Mikic
Predictive Science Inc.
www.predsci.com

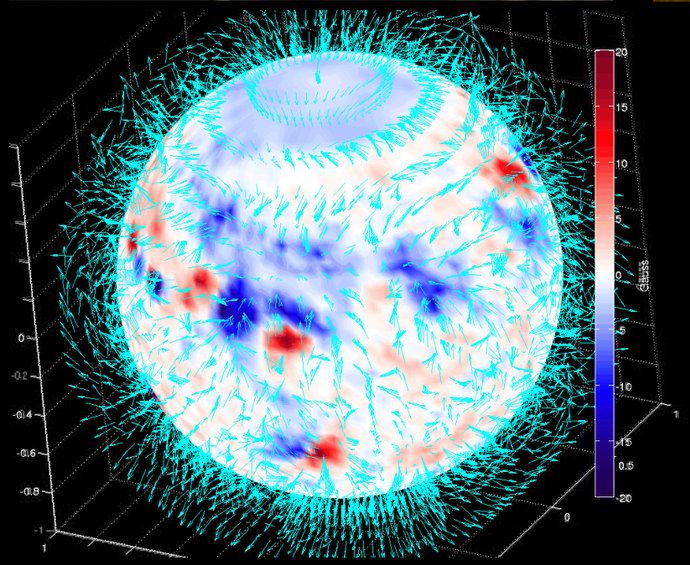
Outline

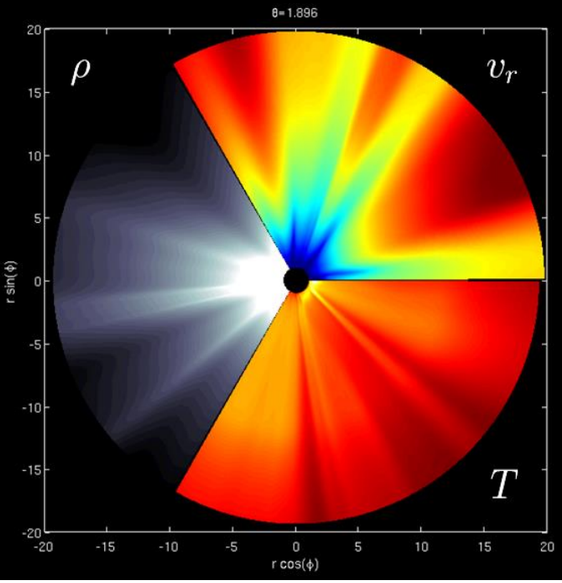
- Potential Field Solutions of the Solar Corona
- Preconditioned Conjugate Gradient
- POT3D
- MPI → MPI+OpenACC
- Test Cases
- Performance Results
- Performance Portability

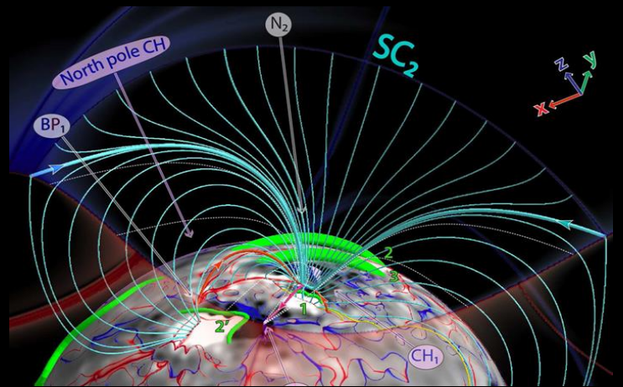


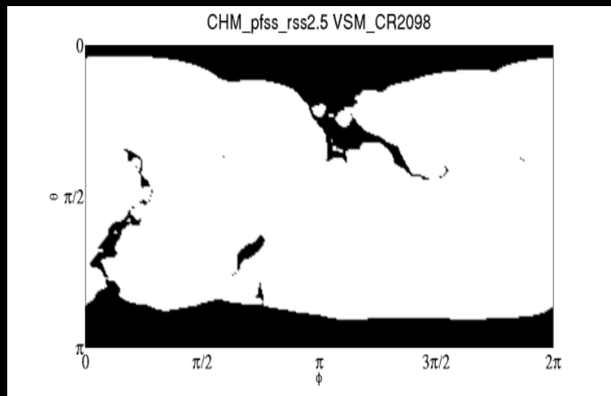
Potential Field Solutions of the Solar Corona

- Studying the Sun's activity important for space weather forecasting
- Knowing the magnetic field is critical
- Available observations: Earth-facing 2D field at solar surface
- Need 3D global field
  - Magnetohydrodynamic simulations [SLOW! ~ days]
  - Potential field (PF) solution [FASTER! ~ minutes]
- PF solutions useful for:
  - Estimating magnetic topology and IMF levels
  - Long-time statistical analysis
  - Providing boundary/initial conditions for modeling solar storms with MHD simulations









Potential Field Solutions of the Solar Corona

Maxwell equation: $\nabla \times \vec{B} = \mu_0 \vec{J}$

Assume no current: $\vec{J} \rightarrow 0 \Rightarrow \nabla \times \vec{B} = 0$

Solution form: $\vec{B} = \nabla \Phi$

Divergence-free condition: $\nabla \cdot \vec{B} = 0$

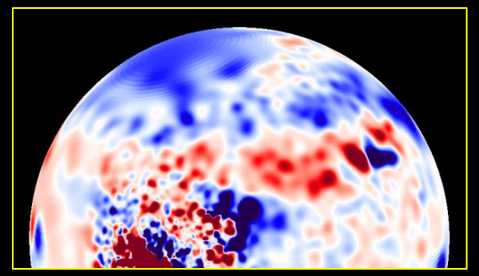
$$\boxed{\nabla^2 \Phi = 0}$$

Laplace equation

Boundary Conditions

$r = R_\odot$ Observed Surface

$$\left.\frac{\partial \Phi}{\partial r}\right|_{R_\odot} = \left. B_r \right|_{R_\odot}$$

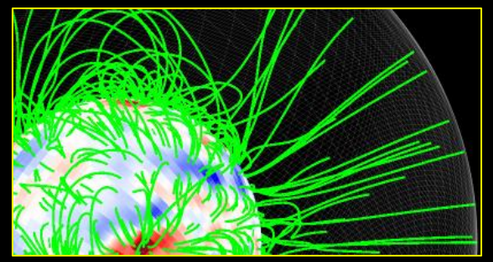


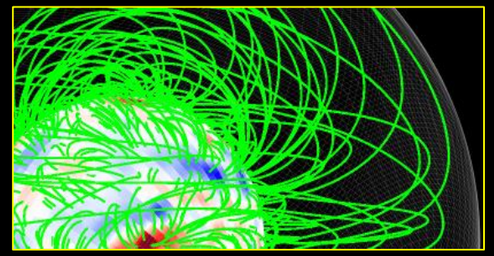
$r = R_1$ (a) Source Surface

$$\Phi|_{R_1} = 0$$

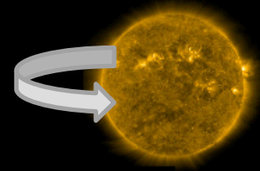
(b) Closed Field

$$\left.\frac{\partial \Phi}{\partial r}\right|_{R_1} = \left. B_r \right|_{R_1} = 0$$





$\phi = 0, 2\pi$ Periodic $\Phi|_{\phi=0} = \Phi|_{\phi=2\pi}$



$\theta = 0, \pi$ Polar average

$$\Phi|_{\theta_0} = \frac{1}{2\pi} \int_{\phi=0}^{\phi=2\pi} \Phi|_{\theta_0 \pm \epsilon} \, d\phi$$

Preconditioned Conjugate Gradient

- PCG: common linear solver for large symmetric sparse matrices
- A `preconditioner' is applied each iteration to improve/ensure convergence

$$\mathbf{A}\,\vec{x} = \vec{b} \qquad (\mathbf{A}\,\Phi = 0)$$

$$\downarrow$$

$$\mathbf{P}^{-1}\,\mathbf{A}\,\vec{x} = \mathbf{P}^{-1}\vec{b} \quad (\mathbf{P} \approx \mathbf{A})$$

- We use two *communication free* preconditioning options:

PC1

Point-Jacobi/Diagonal scaling **(DIAG)**

Cheap, not very effective

PC2

Non-overlapping domain decomposition zero-fill incomplete LU factorization **(ILU0)**

Expensive, much more effective!

Why use/test **PC1** at all?

- For "easy" solves, **PC1** can be faster than **PC2**
- $N_{\text{processors}} \rightarrow N_{\text{grid}}$ ⇒ PC2 → PC1

Preconditioned Conjugate Gradient

Key components of the PCG algorithm

$$\mathbf{P} = \mathrm{pc}()$$

Initialization of PC

- Expensive, but only done once

$$\vec{y} = \mathbf{A}\,\vec{x}$$

Matrix-Vector Product

- Vectorization-friendly
- Point-to-point MPI communication (good scaling)

$$\vec{z} = a\,\vec{x} + b\,\vec{y}$$

Vector-Vector Operations

- Vectorization-friendly

$$r = \vec{x} \cdot \vec{y}$$

Dot-Products

- Vectorization friendly
- Global MPI communication & synchronization (reduced scaling)

$$\vec{y} = \mathbf{P}^{-1}\,\vec{x}$$

Application of PC

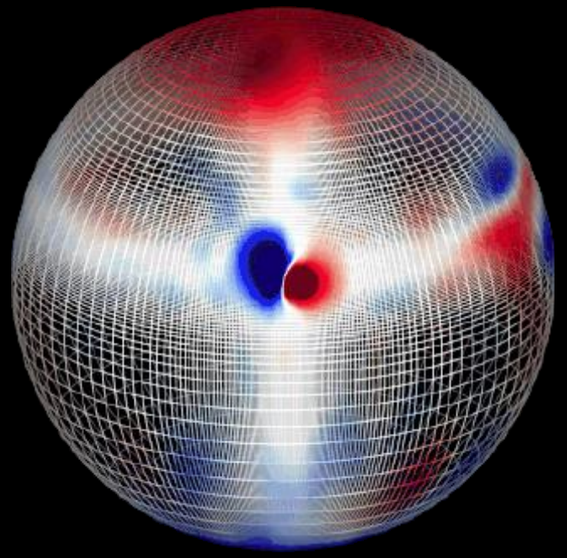
- **PC1:** Vector-Operation (vectorization-friendly)
- **PC2**: Triangular Solves (vectorization-unfriendly)

POT3D Code

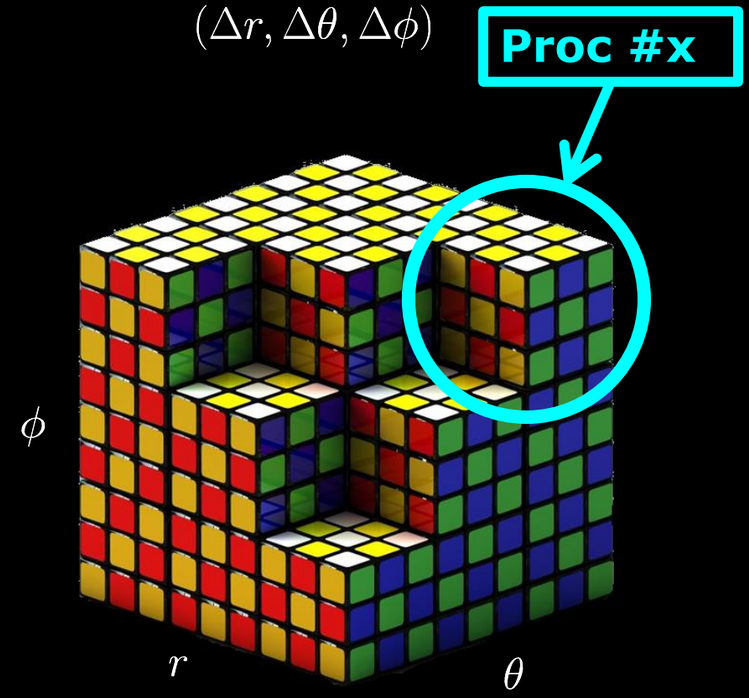
- Written in FORTRAN90, parallelized with MPI
- Finite difference on logically rectangular non-uniform spherical grid

$$\nabla^2\Phi_{i,j,k} \approx \frac{1}{\Delta r_i}\left[\frac{\Phi_{i+1,j,k}-\Phi_{i,j,k}}{\Delta r_{i+\frac{1}{2}}} - \frac{\Phi_{i,j,k}-\Phi_{i-1,j,k}}{\Delta r_{i-\frac{1}{2}}}\right] + \frac{1}{\sin\theta_j\,\Delta\theta_j}\left[\sin\theta_{i,j+\frac{1}{2}}\frac{\Phi_{i,j+1,k}-\Phi_{i,j,k}}{\Delta\theta_{j+\frac{1}{2}}} - \sin\theta_{i,j-\frac{1}{2}}\frac{\Phi_{i,j,k}-\Phi_{i,j-1,k}}{\Delta\theta_{j-\frac{1}{2}}}\right] + \frac{1}{\sin^2\theta_j\,\Delta\phi_k}\left[\frac{\Phi_{i,j,k+1}-\Phi_{i,j,k}}{\Delta\phi_{k+\frac{1}{2}}} - \frac{\Phi_{i,j,k}-\Phi_{i,j,k-1}}{\Delta\phi_{k-\frac{1}{2}}}\right]$$

- 3D Logical Domain Decomposition [MPI_Cart_create()]
- Operator matrix stored in **DIA** format (boundary conditions matrix-free)
- **PC2** preconditioner stored in **CSR** format



$(\Delta r, \Delta\theta, \Delta\phi)$



MPI to MPI+OpenACC – Why?

1) Want to run "in-house"
   a) Off-the-shelf desktops: 2-4 GPUs
   b) HPC single-node rack: 8-16 GPUs
2) Use less HPC allocation $$$
3) Performance Boost:
   Given allocation limits, can GPU code be faster?
4) Stepping stone:
   Our thermodynamic 3D MHD code
   has up to 80% run-time in PCG solvers

Want to do all this in a *portable*, *single-source* way!

-OK!

PGI

As we go along, watch out for:

Difficulties encountered

Things to be careful of!

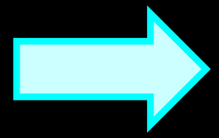
MPI to MPI+OpenACC

Unstructured data regions

Essential* and easy!

```
module fields
  real(r_typ), dimension(:,:,:), allocatable :: phi
end module
```

```
allocate ( phi (nr,nt,np) )
phi = 0.
      [CPU sets initial values of phi…]
!$acc enter data copyin (phi)
```



Compute with phi in OpenACC regions using "present" clause…..

```
!$acc update self (phi)
      [Areas where CPU needs phi (e.g. I/O)]
```

```
!$acc exit data delete(phi)
deallocate (phi)
```

MPI to MPI+OpenACC

FORTRAN array operations

Need to use “kernels” or expand into loops

```
!$acc kernels present(x,p,ap,r)
  x=x+alphai*p
  r=r-alphai*ap
  ap=r
!$acc end kernels
```

```
 4440, Generating present(ap(:),p(:),r(:),x(:))
  4441, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
       4441, !$acc loop gang, vector(128) !
blockidx%x threadidx%x
```

In current OpenACC 2.5, no control on things like vector length….

```
!$acc kernels present(x,br0)
  x( 1,2:ntm1,2:npm1)= x(2,2:ntm1,2:npm1)
 &           -vmask*br0(2:ntm1,2:npm1)*dr1
!$acc end kernels
```

```
 6096, Generating present(br0(:,:),x(:,:,:))
  6097, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
       6097, !$acc loop gang, vector(32) !
blockidx%x threadidx%x
            !$acc loop gang, vector(4) !
blockidx%y threadidx%y
```

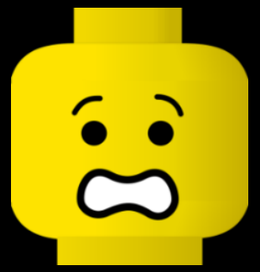
OpenACC 2.6 draft:

Add "num_gangs", "num_workers", and "vector_length“ to “kernels”!

MPI to MPI+OpenACC

Multiple devices

The simplest method:
1 GPU per MPI rank
How easy?



OpenACC with multiple devices is *NOT* that scary …
… even on a large cluster using MPI!

How about in three lines!

```
ngpus_per_node = 4
igpu = MODULO(iprocw, ngpus_per_node)
!$acc set device_num(igpu)
```

Make sure to submit job with
-ntasks-per-node=ngpus_per_node
(or equivalent)

Assumes linear affinity of MPI ranks!
To be safe, should set custom host_file
or mpiexec CPU bindings manually….

PGI 16.10: ~~!$acc set device_num~~
Had to use non-portable API calls

```
use openacc
typ = acc_get_device_type
call acc_init(typ)
igpu = MODULO(iprocw,ngpus_per_node)
call acc_set_device_num(igpu,typ)
```

PGI >17.1: !$acc set device_num supported!

MPI to MPI+OpenACC

CUDA-aware MPI

- MPI finite difference communication involves loading buffers to send data to neighboring processors
- Do *NOT* want buffers filled on CPU!
- CUDA-aware MPI: Send device pointers directly to MPI API and "it just works"! ☺
- This allows use of GPUdirect/NVlink!

Not all HPC system's MPI libraries up-to-date and/or fully tested with CUDA-aware MPI

```
!$acc kernels present(a,sbuf11,sbuf12)
        sbuf11(:,:)=a(n1-1,:,:)
        sbuf12(:,:)=a(   2,:,:)
!$acc end kernels
```

LOAD BUFFERS ON GPU

```
!$acc host_data use_device(sbuf11,sbuf12,rbuf11,rbuf12)
        call MPI_Isend (sbuf11,lbuf,ntype_real,iproc_rp,tag,
     &                  comm_all,reqs(1),ierr)
        call MPI_Isend (sbuf12,lbuf,ntype_real,iproc_rm,tag,
     &                  comm_all,reqs(2),ierr)
        call MPI_Irecv (rbuf11,lbuf,ntype_real,iproc_rm,tag,
     &                  comm_all,reqs(3),ierr)
        call MPI_Irecv (rbuf12,lbuf,ntype_real,iproc_rp,tag,
     &                  comm_all,reqs(4),ierr)
        call MPI_Waitall (4,reqs,MPI_STATUSES_IGNORE,ierr)
!$acc end host_data
```

```
        if (iproc_rm.ne.MPI_PROC_NULL) then
!$acc kernels present(a,rbuf11)
          a( 1,:,:)=rbuf11(:,:)
!$acc end kernels
        end if
        if (iproc_rp.ne.MPI_PROC_NULL) then
!$acc kernels present(a,rbuf12)
          a(n1,:,:)=rbuf12(:,:)
!$acc end kernels
        end if
```

UNLOAD BUFFERS ON GPU

MPI to MPI+OpenACC

Calling cuSparse from FORTRAN (PC2 only!)

- PC2: Requires triangular solves (vector-unfriendly)
- Advanced algorithms difficult to implement… Use freely-available libraries! **cuSparse**!
- Portability: Call C code from FORTRAN (instead of FORTRAN bindings)
- For compiler-independent, "correct" way, see: "*Parallel Programming in OpenACC*" pg. 283

lusol.c

```c
void lusol_cusparse(double* x, double* CSR_LU,
                    int* CSR_I,int* CSR_J, int N, int M)
{  // Forward solve (Lx=y)
  cusparseDcsrsv_solve(cusparseHandle,
          CUSPARSE_OPERATION_NON_TRANSPOSE, N, &one, L_described,
                       CSR_LU, CSR_I, CSR_J, L_analyzed, x, x);
  // Backward solve (Uy=x)
  cusparseDcsrsv_solve(cusparseHandle,
          CUSPARSE_OPERATION_NON_TRANSPOSE, N, &one, U_described,
                       CSR_LU, CSR_I, CSR_J, U_analyzed, x, x);
  cudaDeviceSynchronize();
}
```

```f90
module cusparse_interface
  interface
    subroutine lusol_cusparse(
&                 x,CSR_LU,CSR_AI,CSR_AJ,N,M)
&                 BIND(C, name="lusol_cusparse")
      use, intrinsic :: iso_c_binding
      integer(C_INT), value :: N
      type(C_PTR), value :: x,CSR_LU,CSR_AI,CSR_AJ
    end subroutine lusol_cusparse
  end interface
end module
```

```f90
use, intrinsic :: iso_c_binding
use cusparse_interface
integer(c_int) :: cN,cM

!$acc host_data use_device(x,a_csr,a_csr_ja,…)
   call lusol_cusparse(C_LOC(x(1)),
  &                    C_LOC(a_csr(1)),
  &                    C_LOC(a_csr_ia(1)),
  &                    C_LOC(a_csr_ja(1)),cN,cM)
!$acc end host_data
```

Compile!

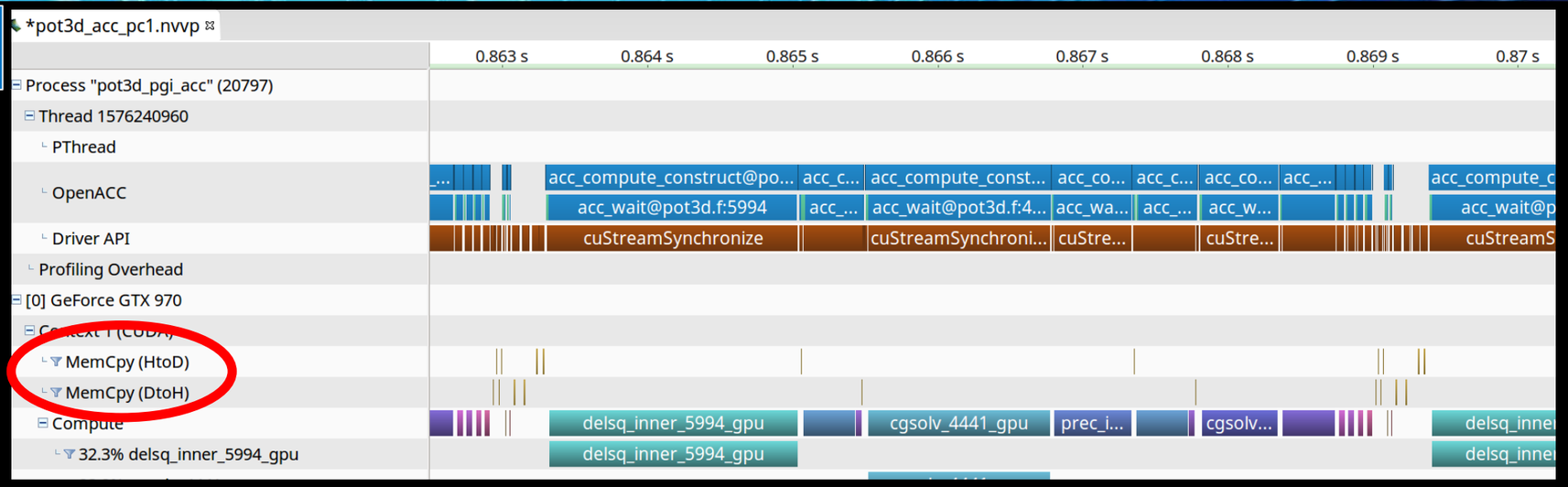
```
nvcc -c [FLAGS] lusol.c
pgf90 [FLAGS] lusol.o [LIBS] pot3d.f
```

MPI to MPI+OpenACC - Profiling

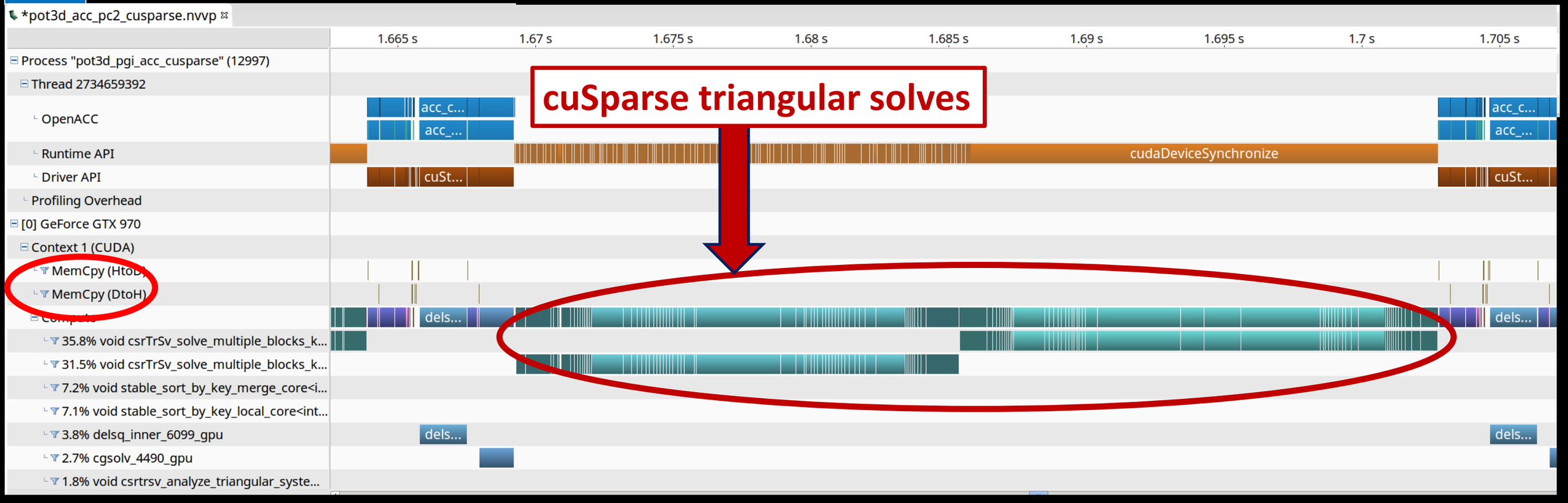
PROFILE!

PGPROF

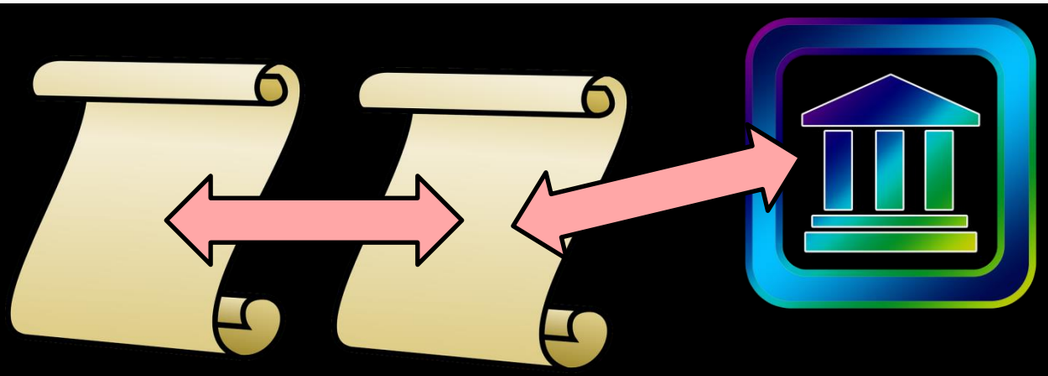
PC1



PC2

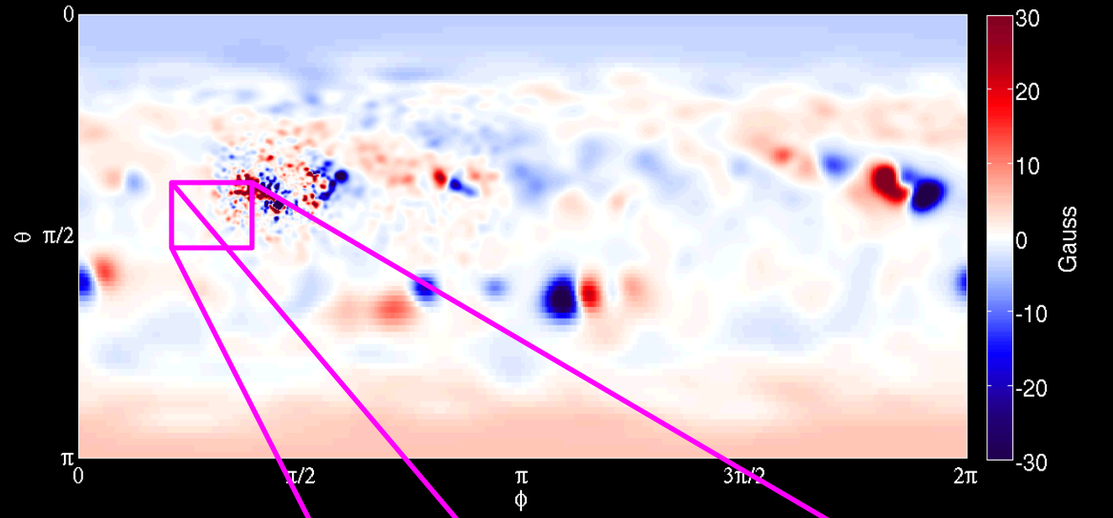


Summary of MPI to MPI+OpenACC

	Modification	Portability
PC1	~1% ~90 lines of comments	 Single source for GPU and CPU!
PC2	~2% ~150 lines of comments and new code	 New source code and library dependency (Could make `portable' with `ifdefs'…)

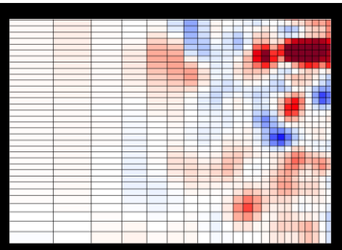
Test Problem

- Real-world test problem
- Boundary magnetic map from HMI[CR2099+Daily(8/1/2010)]
- Used to study sympathetic solar storms [Titov, et al, Ap.J. 759:70 (2012)]



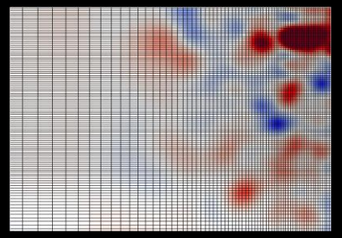
Three problem sizes

Small 120x120x240 ~ 3 million points



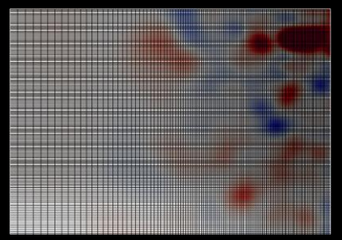
PCG Iterations	
PC1	3883
PC2	0697 → 0874

Medium 100x360x720 ~25 million points

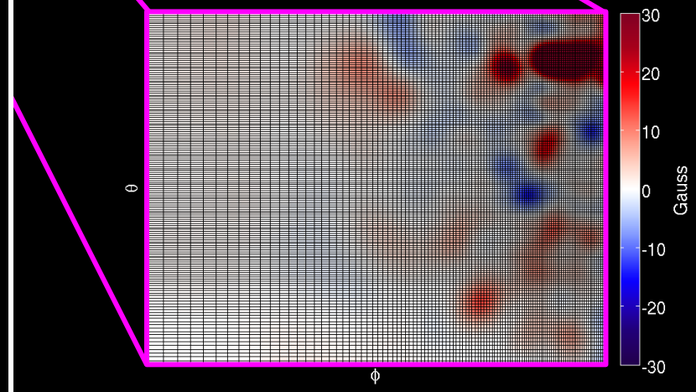


PCG Iterations	
PC1	8719
PC2	1273 → 1871

Large 200x720x1440 ~207 million points



PCG Iterations	
PC1	19995
PC2	2563 → 3221



	N	min Δ	max Δ	max ΔΔ
θ	480	0.1°	2.6°	≈ 15%
ϕ	1151	0.1°	0.9°	≈ 4%

HPC Environment

SYSTEMS

Allocation provided by:

XSEDE
Extreme Science and Engineering Discovery Environment

NVIDIA

NVIDIA PSG Cluster

Allocation provided by:
NVIDIA Coorperation

Puget systems

Local Desktop

HARDWARE SPECS

	COMET NODE	DESKTOP
CPU Model	(2x) Intel Xeon E5-2680v3	Intel Xeon E5-1650v4
Base Clock	2500 MHz	3600 MHz
# Cores	24	6
DP FLOPS	≈ 0.96 TFlop/s	≈ 0.35 TFlop/s
RAM	128 GB	32 GB
Mem Band	68 GB/sec	76.8 GB/s

	COMET NODE	NVPSG NODE
GPU Model	NVIDIA Tesla K80 (x1)	NVIDIA Tesla P100 (PCIe)
Base Clock	560 MHz	1328 MHz
# DP Cores	832	1792
DP FLOPS	≈ 0.93 TFlops	≈ 4.7 TFlops
RAM	12 GB	16 GB
Mem Band	240 GB/sec	549 GB/sec

SOFTWARE

intel
2015.2.164

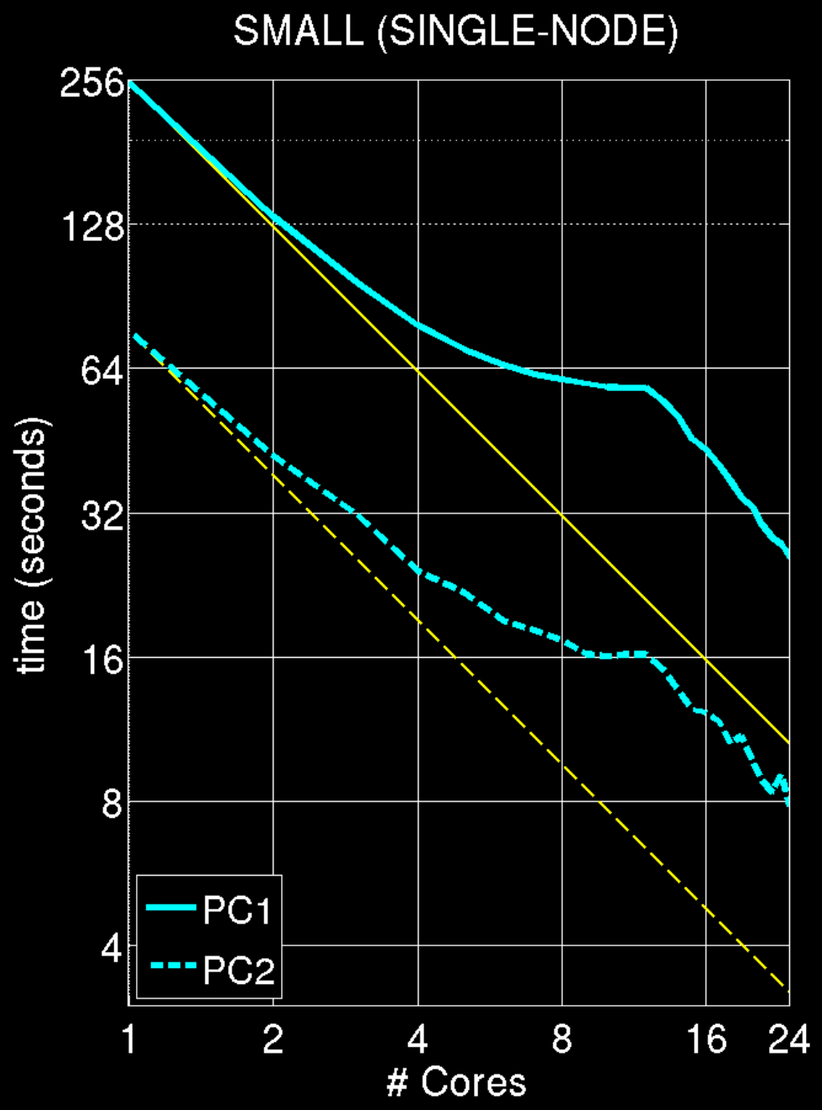
+ MVAPICH
for infiniband v2.1

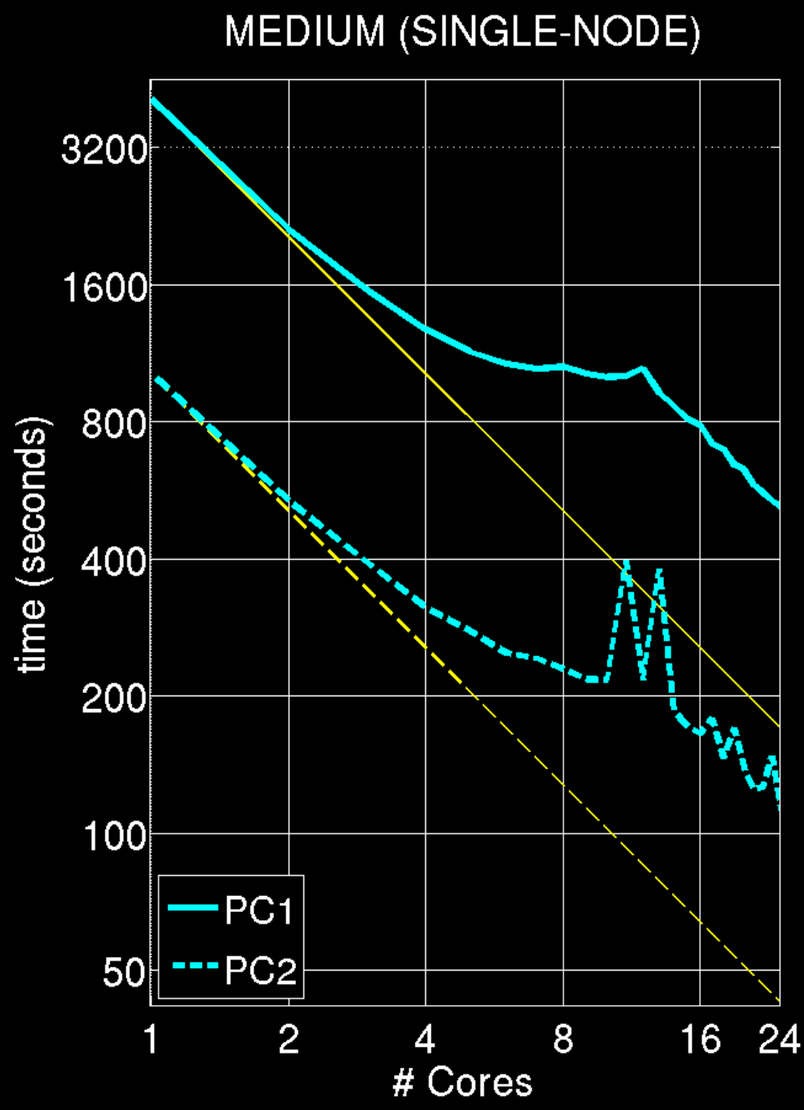
PGI
16.10 + CUDA 8.0

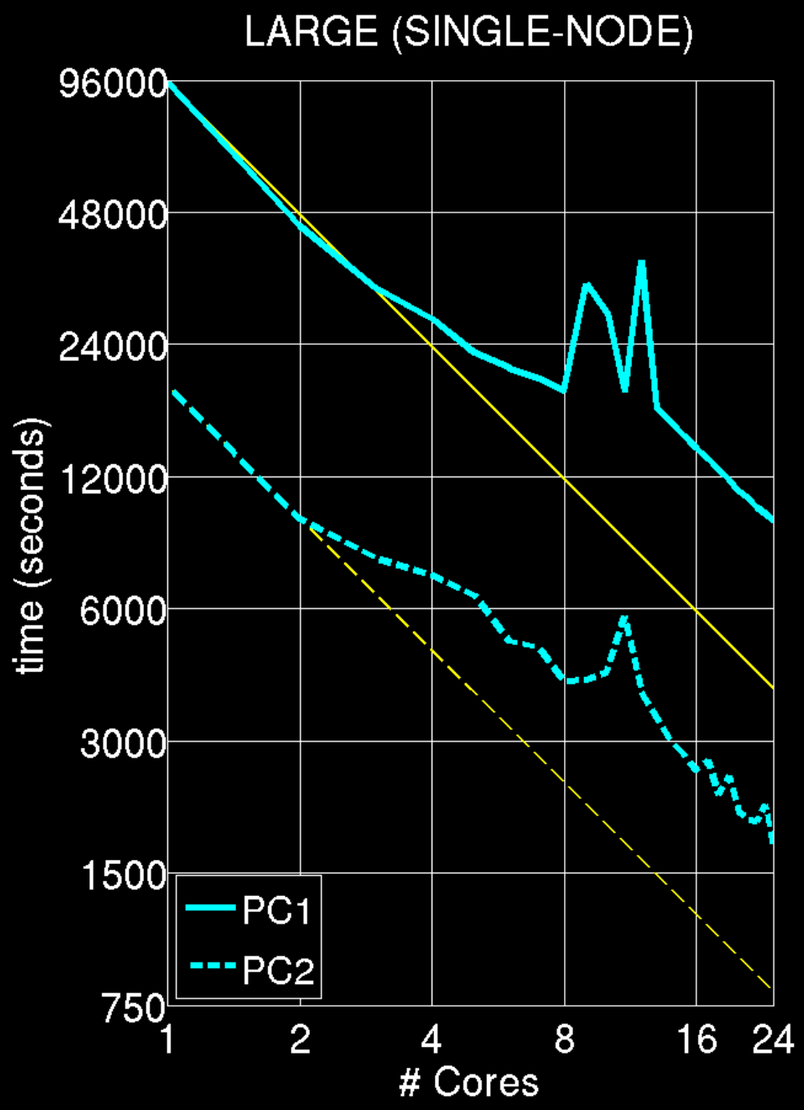
+ OpenMPI
1.10.2

All timing results made with MPI timers for solve and setup (i.e. excludes I/O)

Timings - Original CPU Scaling

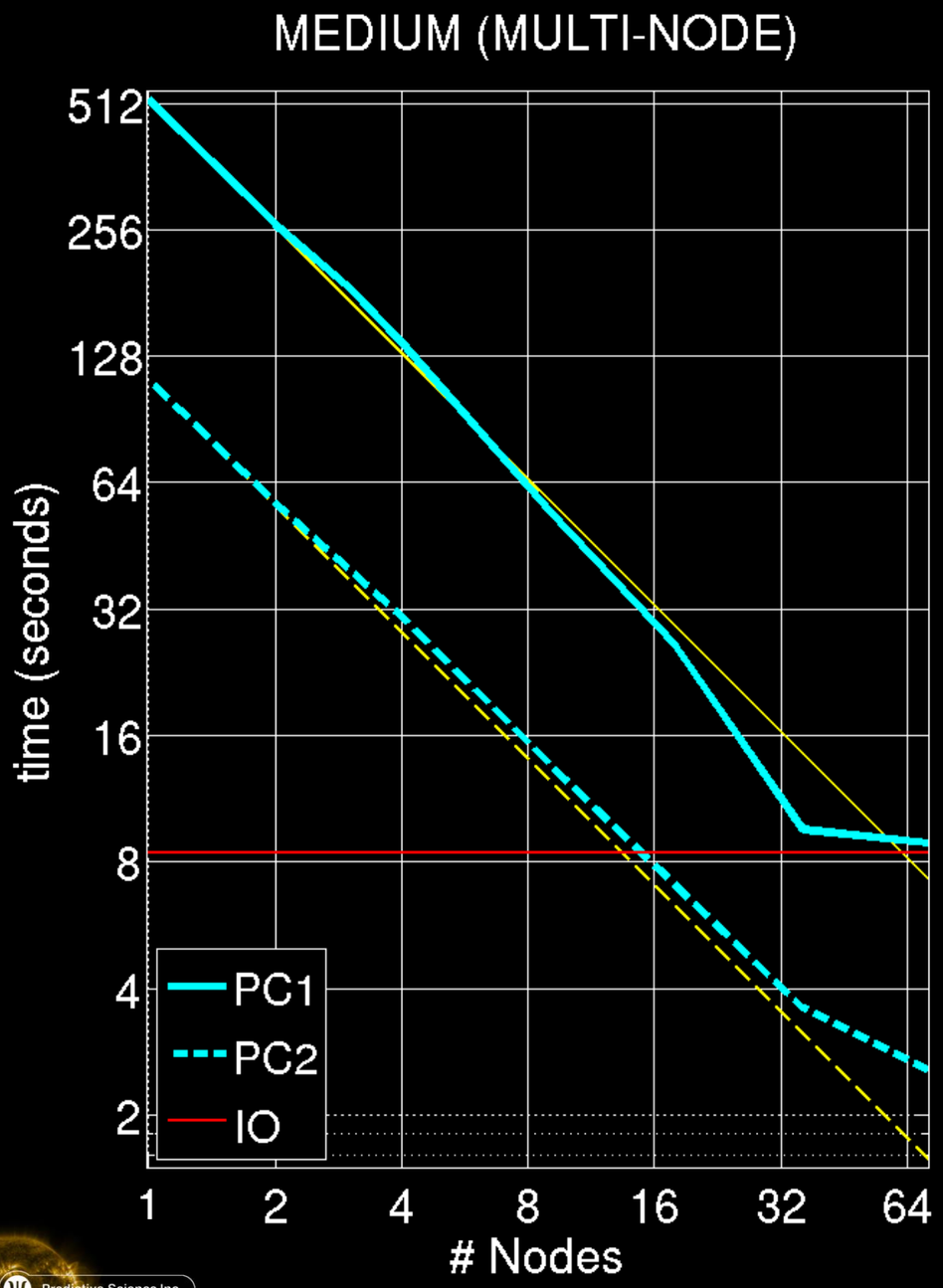


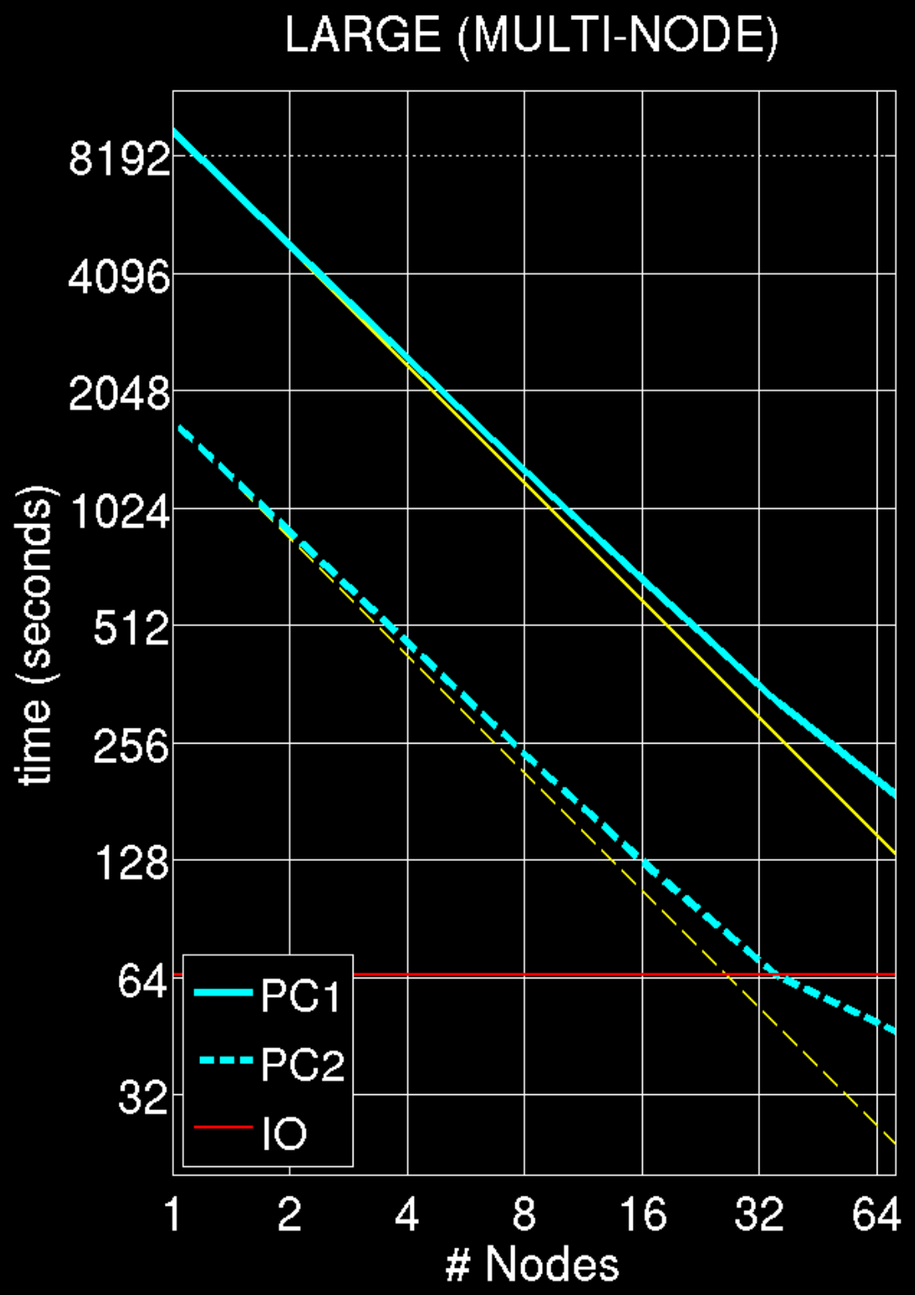




- PC2 more than twice as fast as PC1!
- Strange scaling behavior

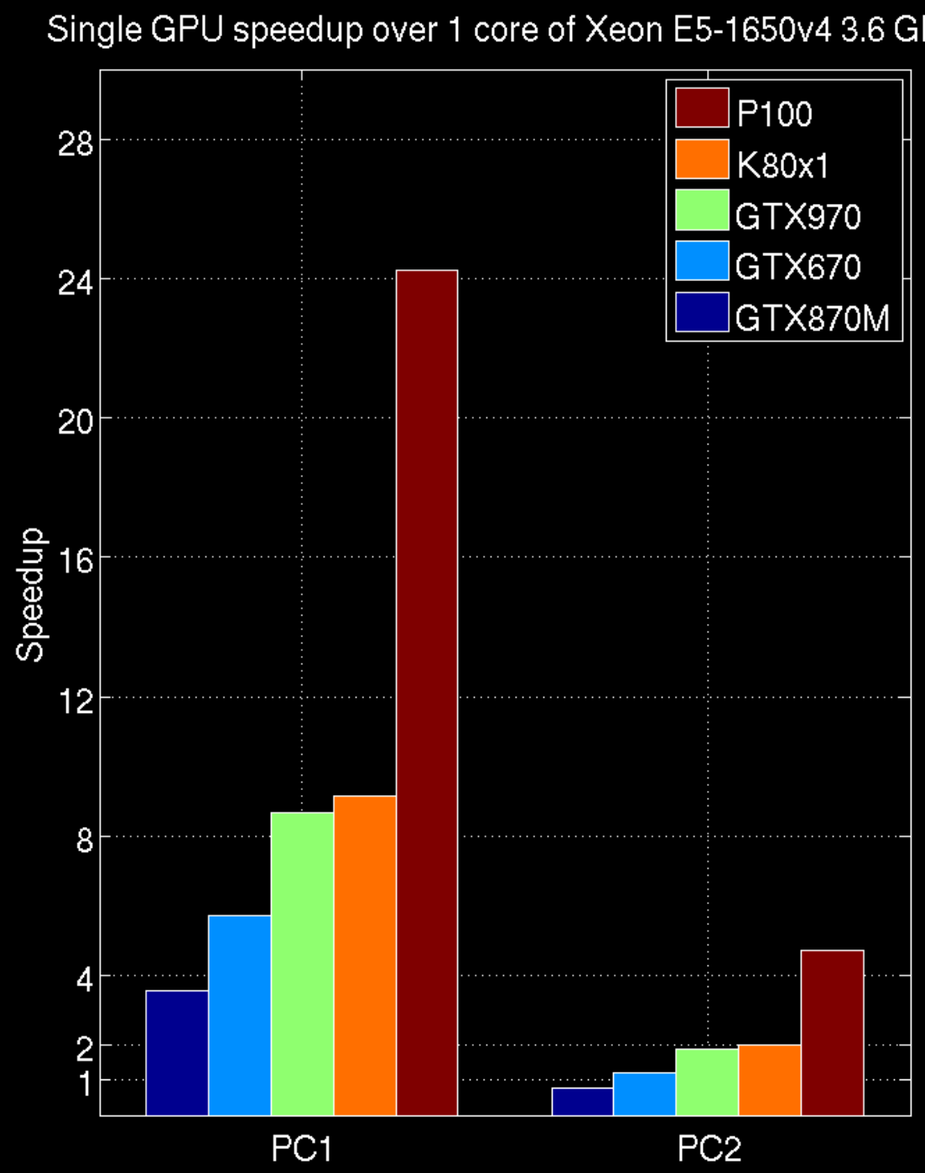
Timings - Original CPU Scaling



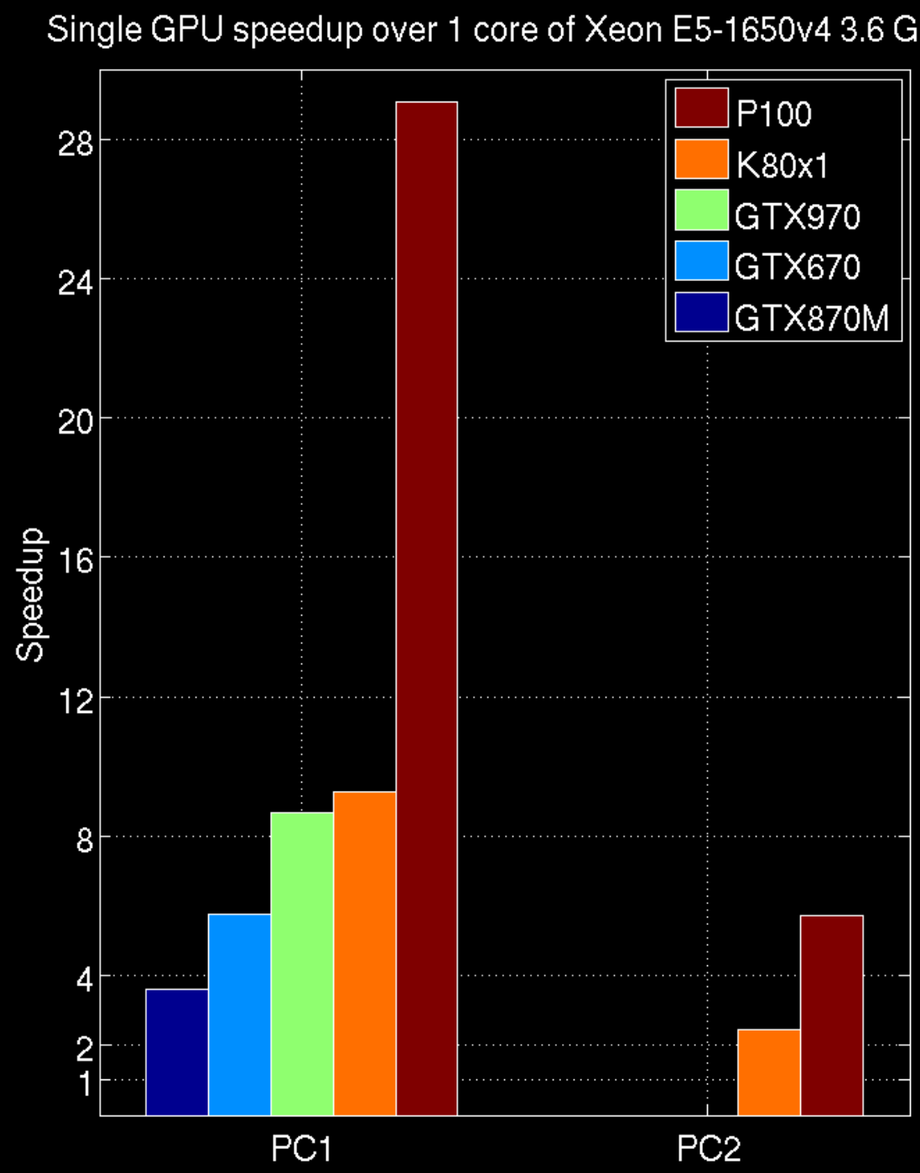


- Multi-node scaling much better than single-node
- Scaling starts to weaken near the 40-50 node level
- Algorithm hits I/O time for largest runs

Timings - Single GPU vs Serial CPU



SMALL

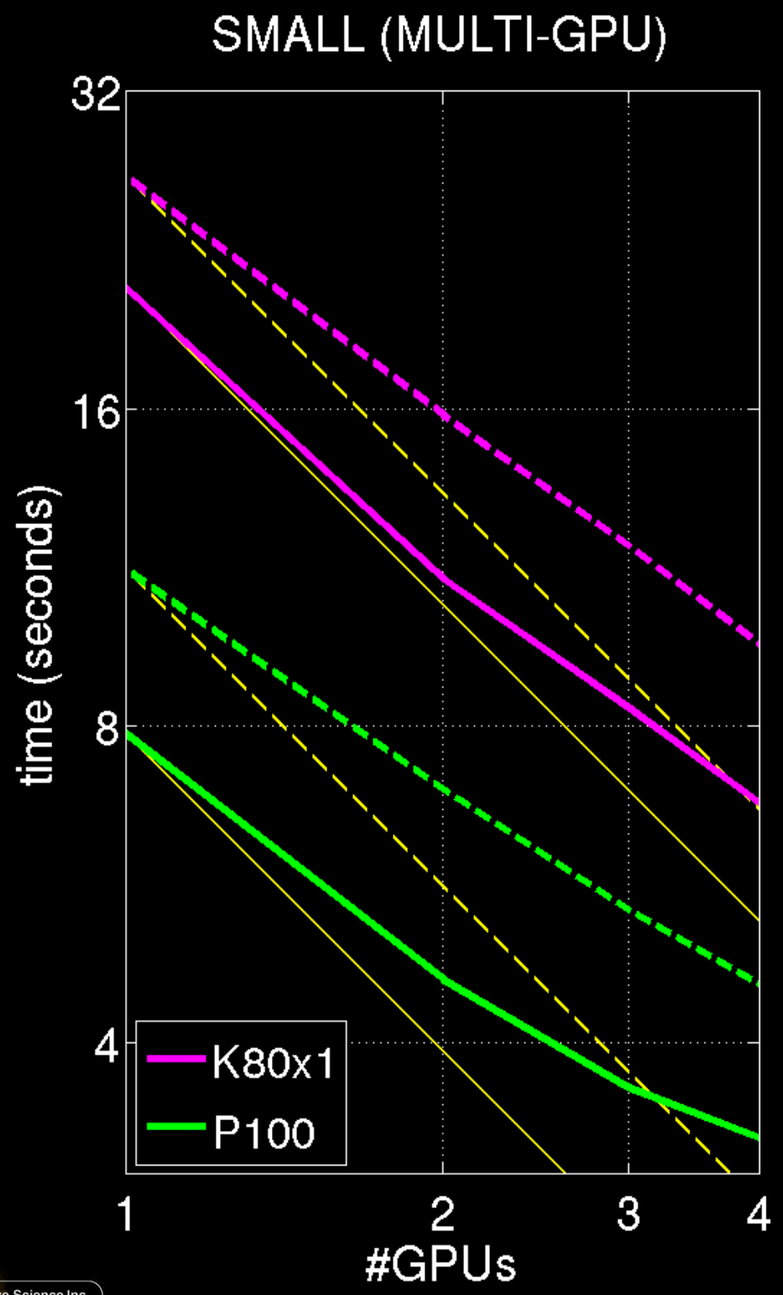


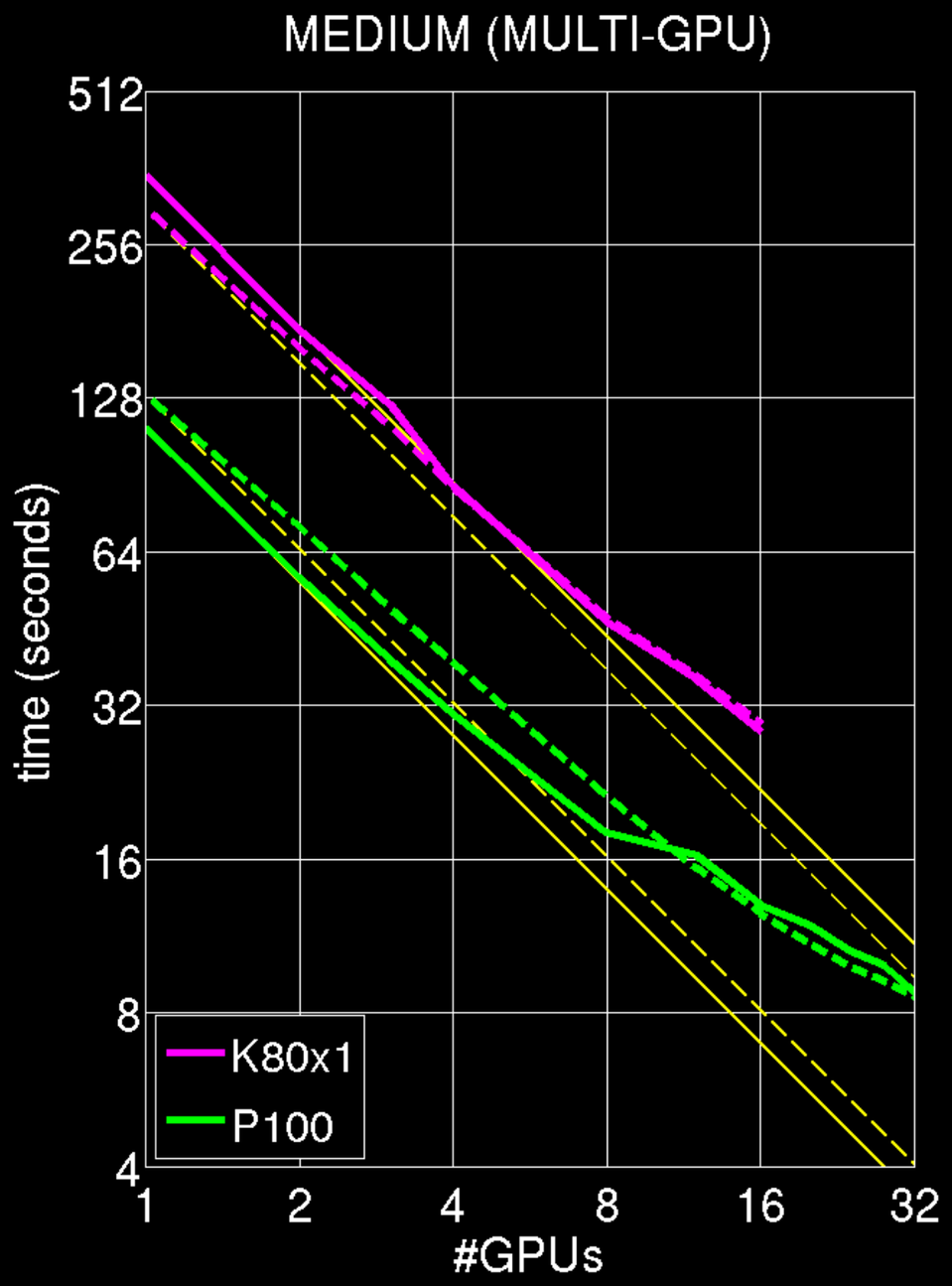
MEDIUM

- PC1 speedup much better than PC2
- Pascal much faster than Kepler

Problem size important!
(must fit on GPU!)

Timings - GPU Scaling

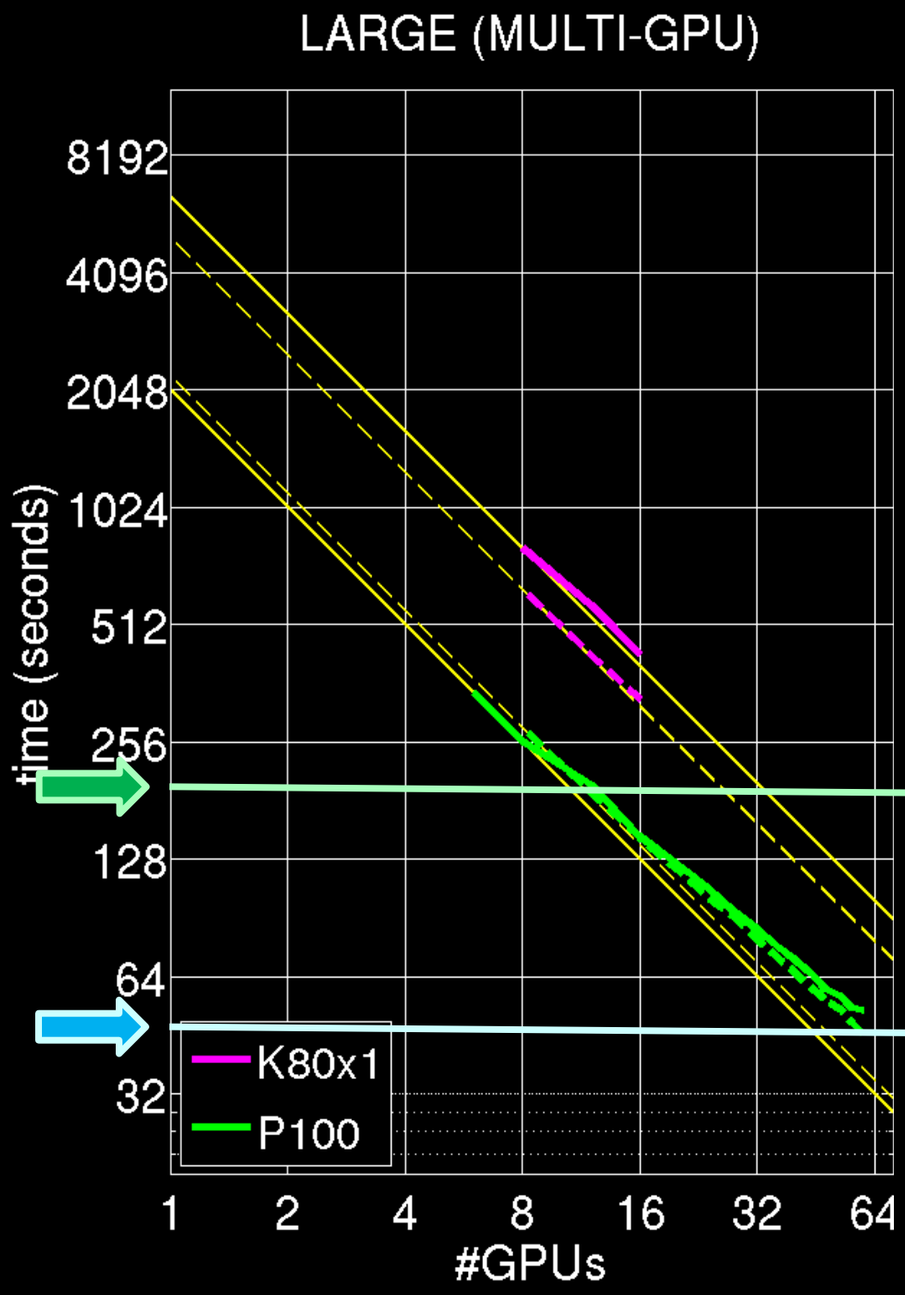




- Pascal more than twice as fast as Kepler
- PC2 slower than PC1 for small run!
- PC1~PC2 for medium run

Timings - GPU Scaling

- Scaling has improved
- P100: PC1~PC2
- PC1: 14-GPU run faster than CPU run on max 72 nodes
- PC2: 60-GPU run faster than CPU run on max 72 nodes

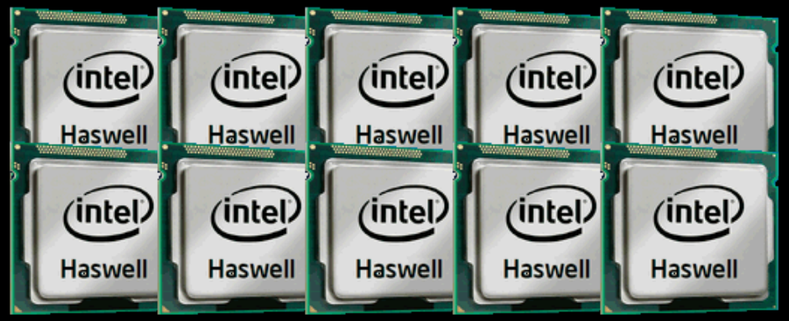


Summary of GPU Performance (Large Run)

PC1

1x P100

≈



5x 2x12-core Haswell

PC2

1x P100

≈

1x 2x12-core Haswell

- Triangular solve performance on GPUs (in this CSR case) needs improvment
- Are there vectorizable preconditioners with comprable performance as ILU?

Is a simple diagonal scaling the best preconditioner for conjugate gradients on supercomputers?

Giorgio Pini and Giuseppe Gambolati

[Adv. Water Resources, 1990, Vol. 13, No. 3]

Summary of MPI+OpenACC POT3D on NVIDIA GPUs

How did we do?

	PC1	PC2
1) Wanted to run "in-house" Desktop with 4x P100s ~ 20x/04x HPC 24-core nodes for PC1/PC2 HPC Node with 16x P100s ~ 80x/16x HPC 24-core nodes for PC1/PC2	👍	👉
2) Wanted to use less HPC allocation With 4x P100s per node, allocation savings of ~ 20x/4x for PC1/PC2	👍	👍
3) Wanted a performance boost PC1: 60x P100s yield 4x speedup over max 72-nodes CPU PC2: 60x P100s yield only slight speedup over max 72-nodes CPU	👍	👉
Wanted to do all this in a *portable*, *single-source* way…	👍	👎

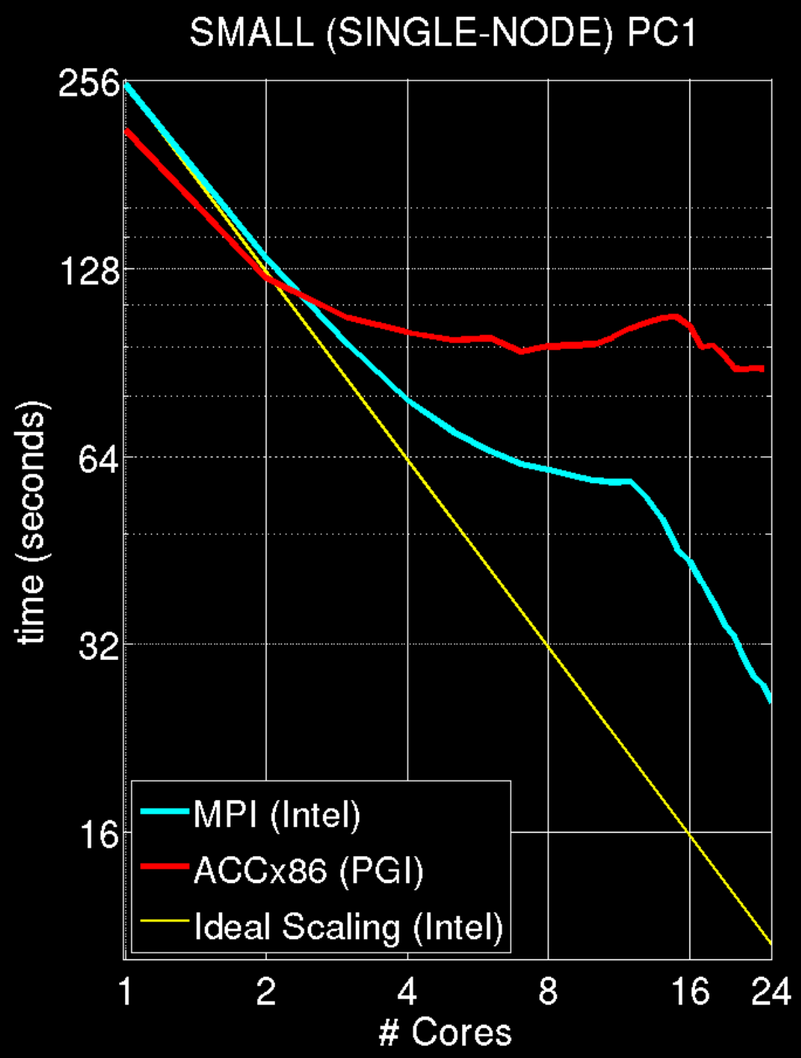
Performance Portability

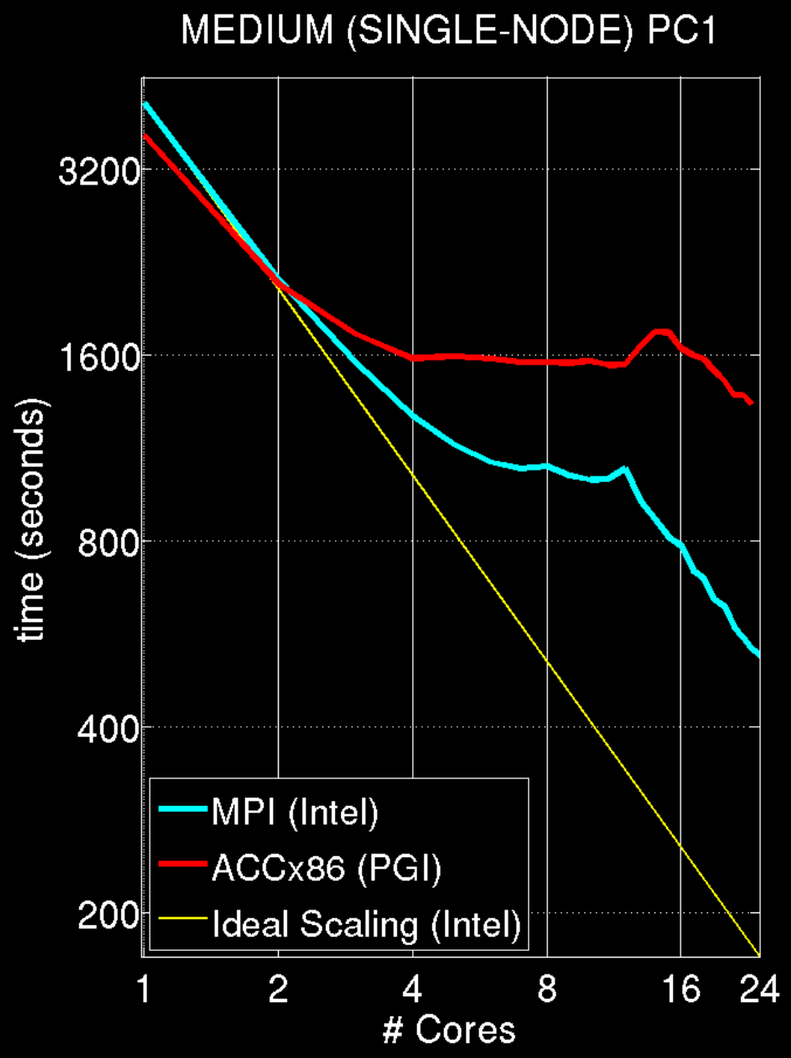
One code to run on them all… efficiently…

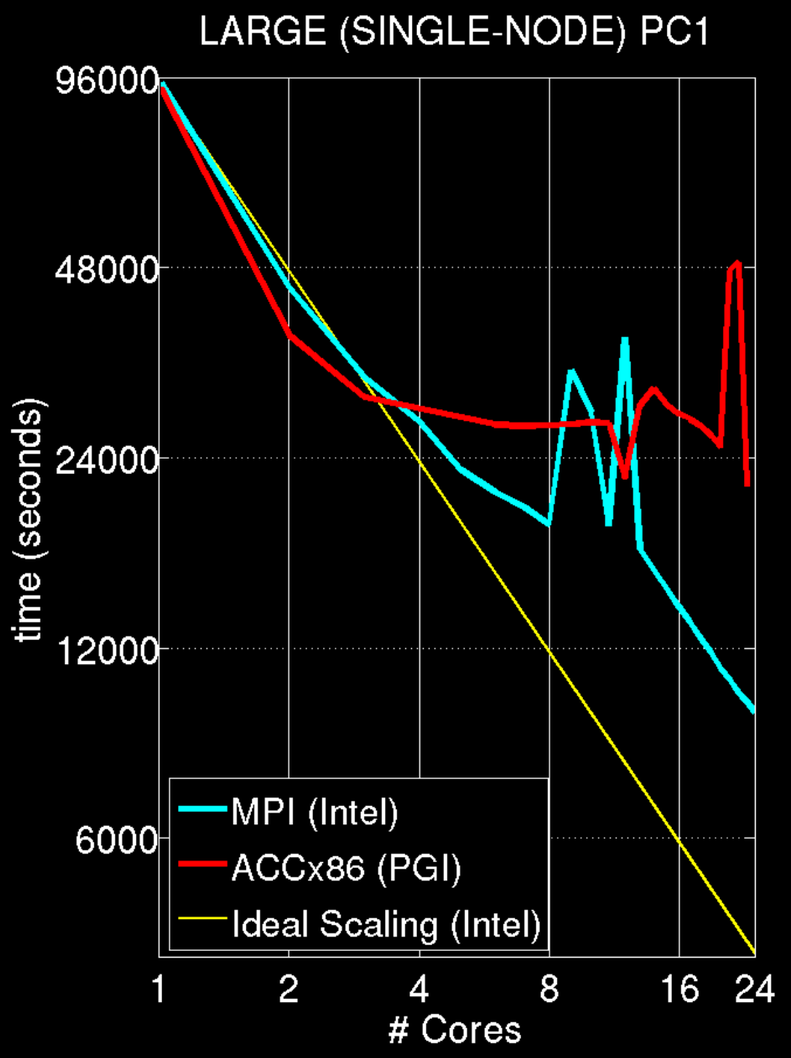
- OpenACC for x86 Multicore
- OpenACC for NVIDIA GPUs
- OpenACC for AMD GPUs
- OpenACC for OpenPower
- OpenACC for Intel KNL

Performance Portability (x86 multicore)

For PC1, with ***no code modifications***, can compile for multi-core x86 CPU (-ta=multicore)



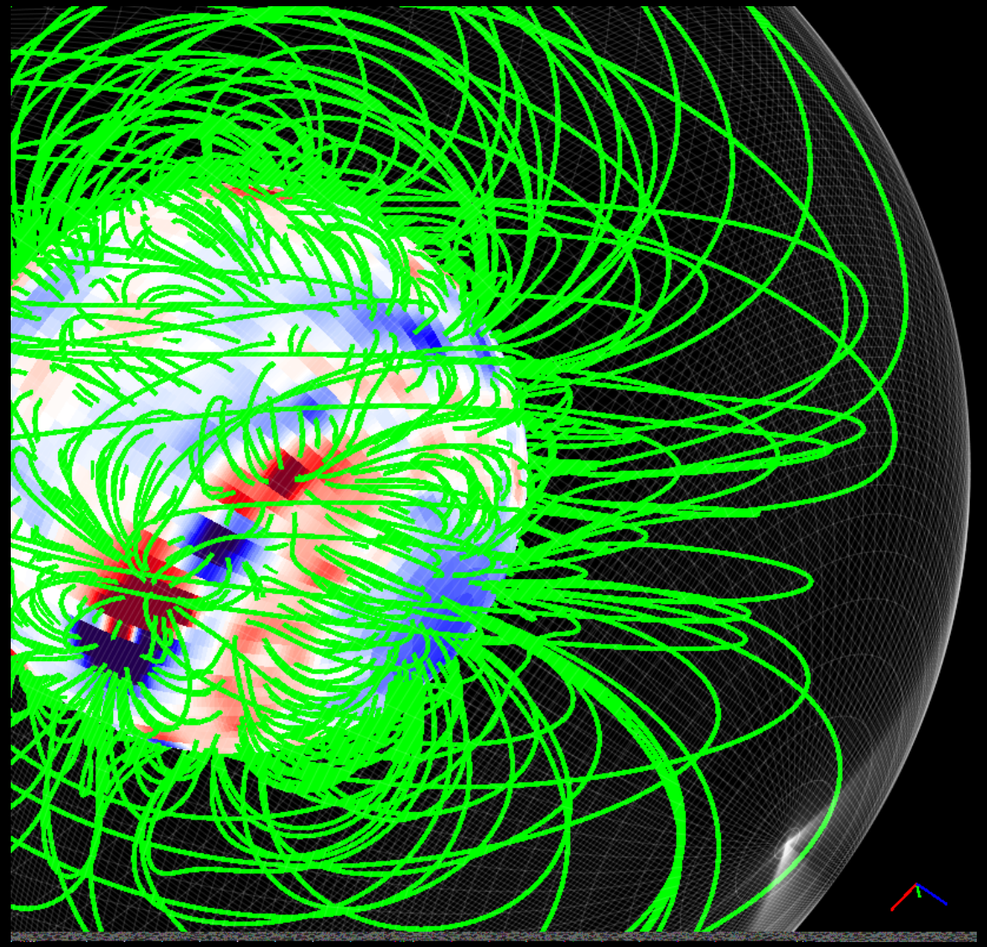




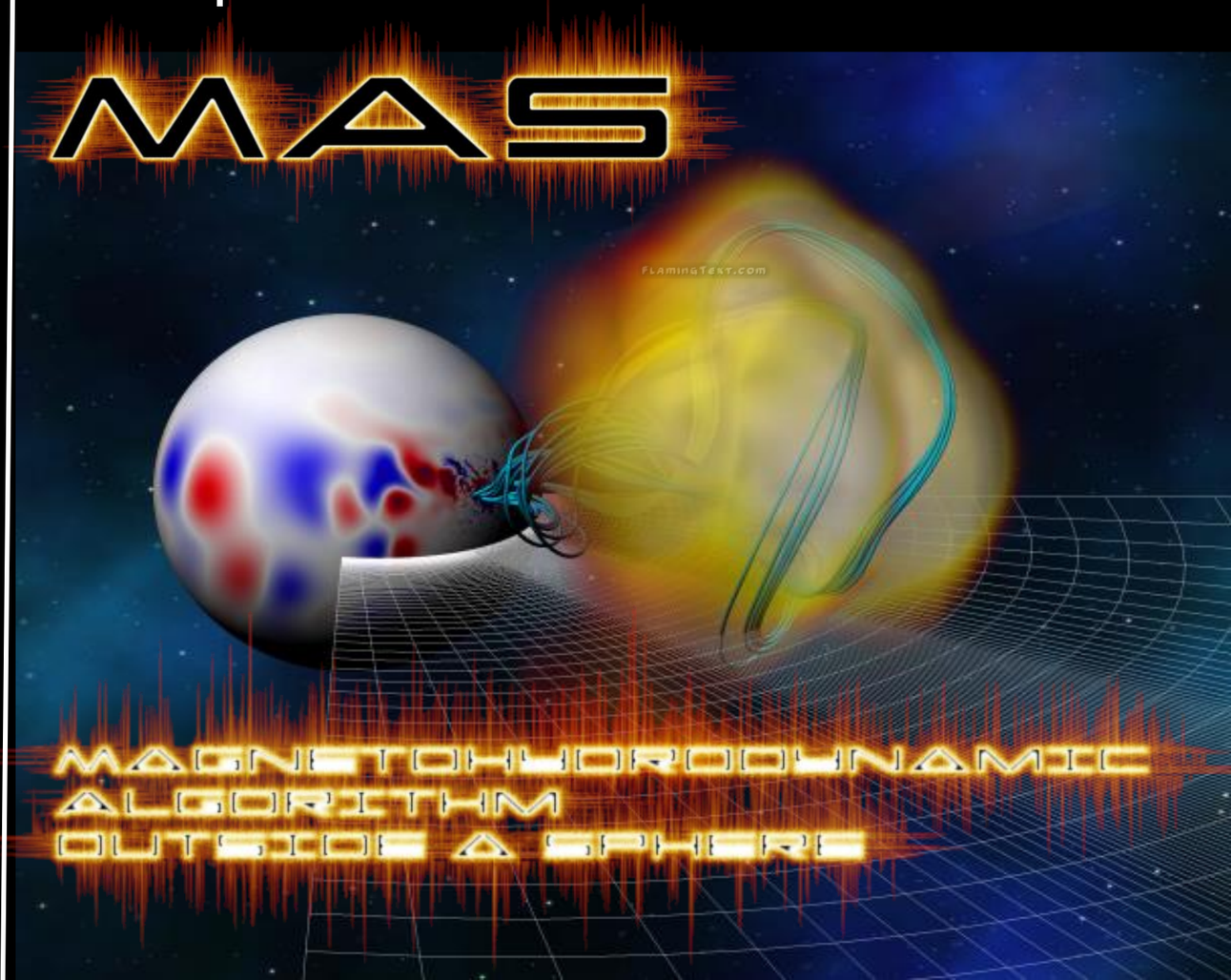
- OpenACC x86 scaling poorer than MPI-only (expected to match OpenMP)

Future Work

- OpenACC POT3D will be used in computing over 20,000 PF solutions covering over 4 years of solar data



- Our global MHD simulation code uses similar PCG solvers, so look for an initial OpenACC implementation at next GTC!



Questions?

OpenACC User Group

Directives for Accelerators —> More Science. Less programming

Twitter @OpenACCorg
Facebook @OpenACCorg
LinkedIn OpenACC Developers

This work was supported by
- *NSF's Frontiers in Earth System Dynamics program*
- *NASA's Living with a Star program (lws.gsfc.nasa.gov)*
- *Air Force Office of Scientific Research*

We gratefully awknowlege NVIDIA Coorperation for donating allocation use of their PSG Cluster for the P100 timings.

NVIDIA GPU TECHNOLOGY CONFERENCE

Predictive Science Inc.