Update

Ross Heikes, C.S. Konor and D. Randall

Dept. of Atmospheric Science
Colorado State University

Office of Science

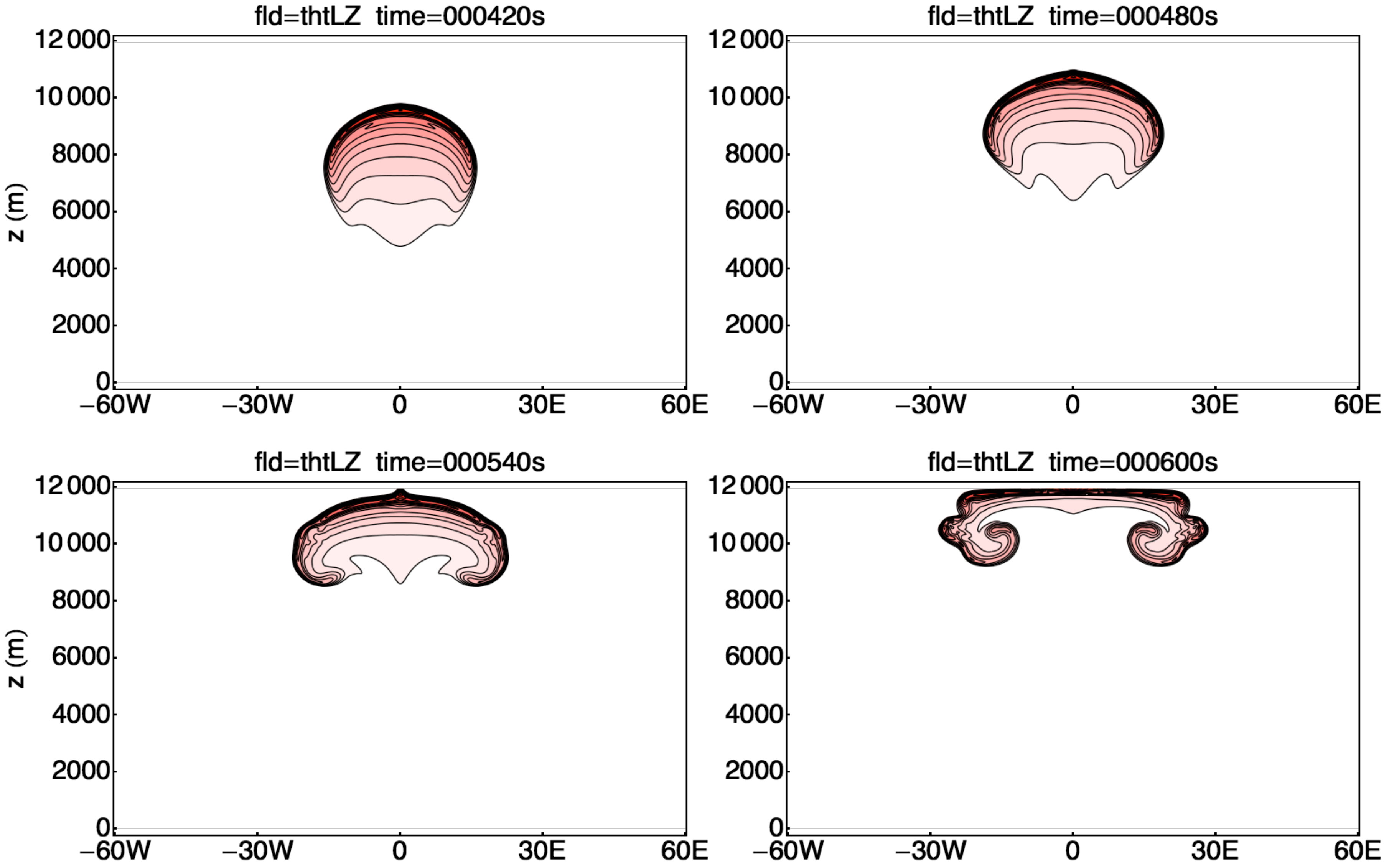
SciDAC
Scientific Discovery through Advanced Computing

CMMAP
Reach for the sky.

Outline

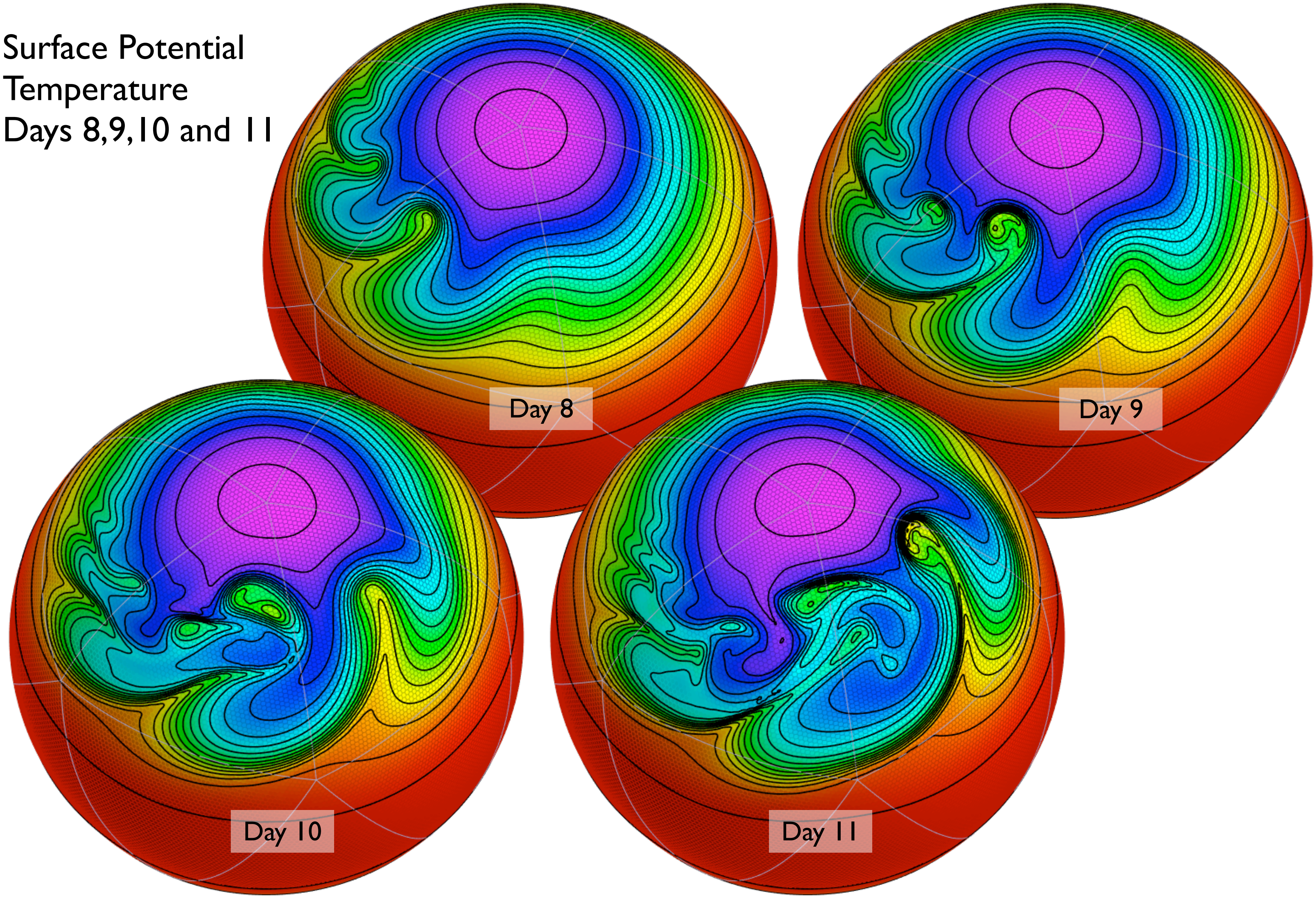
- Status of the Vorticity-Divergence Dynamical Core with the unified system of equations (Arakawa and Konor).
- Update on the continuing grid optimization saga
- First steps of a MPI/OpenMP hybrid model

Warm Bubble Test



Extratropical cyclone

- Surface Potential Temperature
- Days 8,9,10 and 11



Since the last meeting

- Improved efficiency of the multigrid elliptic solvers
- Merging of my code with the SVN repository code. Anyone in the world can check out the unified model.

Grid Optimization Saga

- The **grid optimization algorithm** positions the grid point to improve the convergence rate of the finite-difference operators.
- Number of independent variables is shown in the table.
- Since the last meeting we have tried to extend the optimization to grid 13.
- Grid 13 has proven itself difficult to fit onto any normal computer.

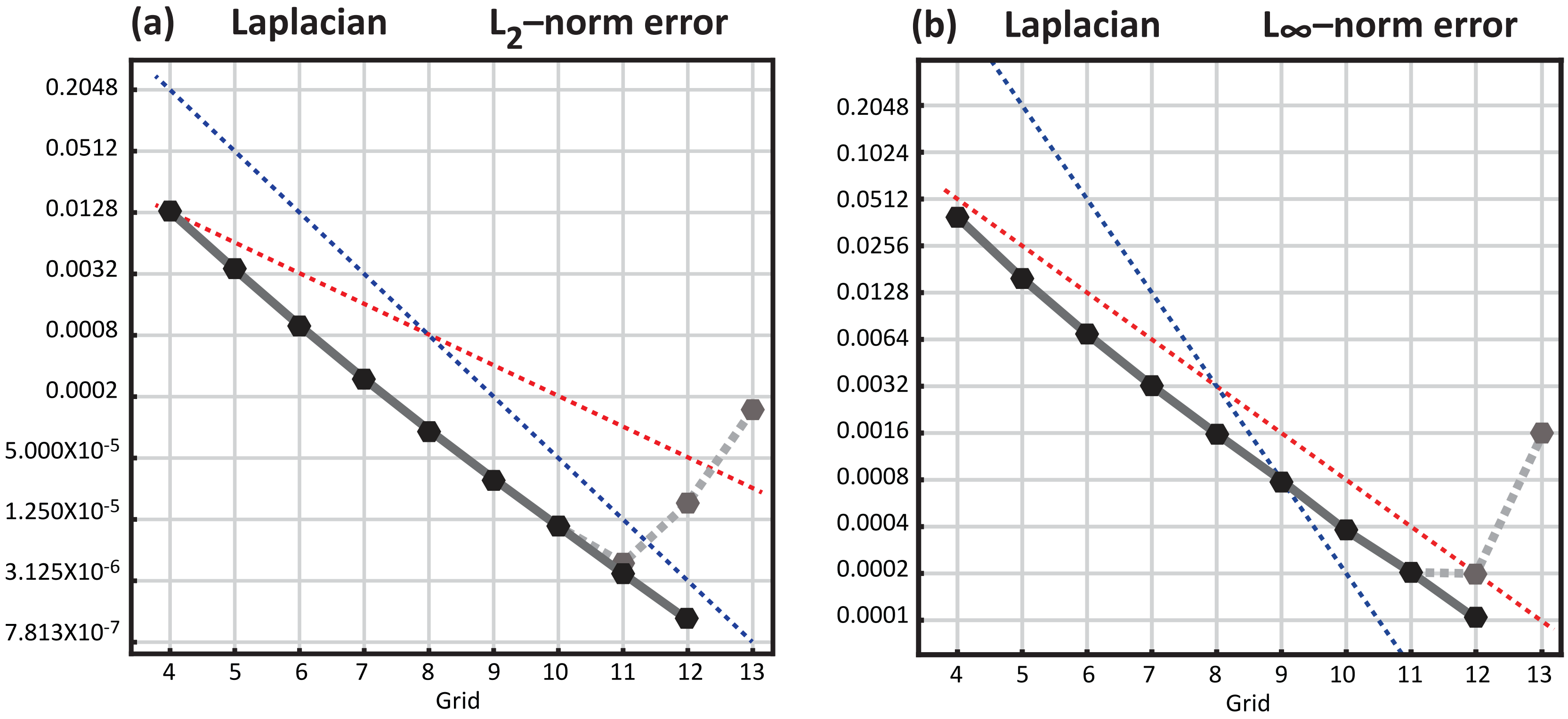
grid resolution	number of independent variables
(9) 2,621,442 (15.64km)	32,768
(10) 10,485,762 (7.819km)	131,072
(11) 41,943,042 (3.909km)	524,288
(12) 167,772,162 (1.955km)	2,097,152
(13) 671,088,642 (0.997km)	8,388,608

Grid Optimization Saga

- These figures show the error (L2-norm error and infinity-norm error) in the Laplace operator as a function of grid number.

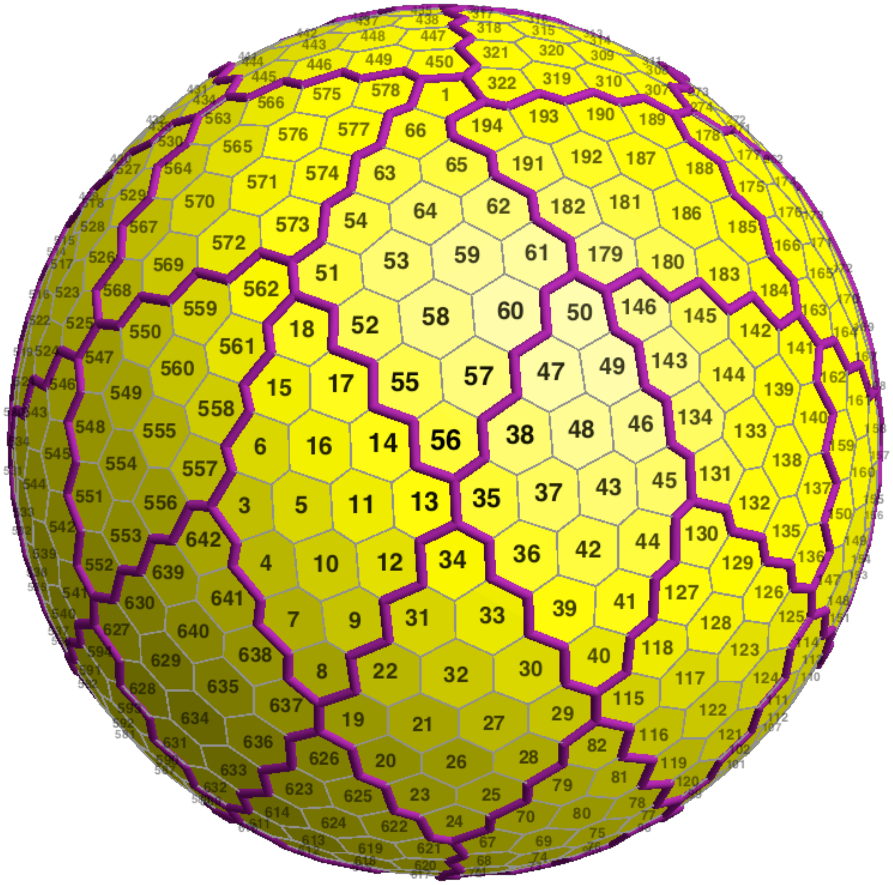
  1) solid line is with 128-bit numbers
  2) dashed line is with 64-bit numbers. This is the extension to grid 13.

- red dashed line shows 1st-order convergence.
  blue dashed line shows 2nd-order convergence.



Parallel domain decomposition

- The global grid is partitioned into subdomain blocks of cells.
- **Blocks are assigned to MPI processes** and boundary information is transmitted between processes with MPI messages.
- For example, 642 cells partitioned into 40 blocks.

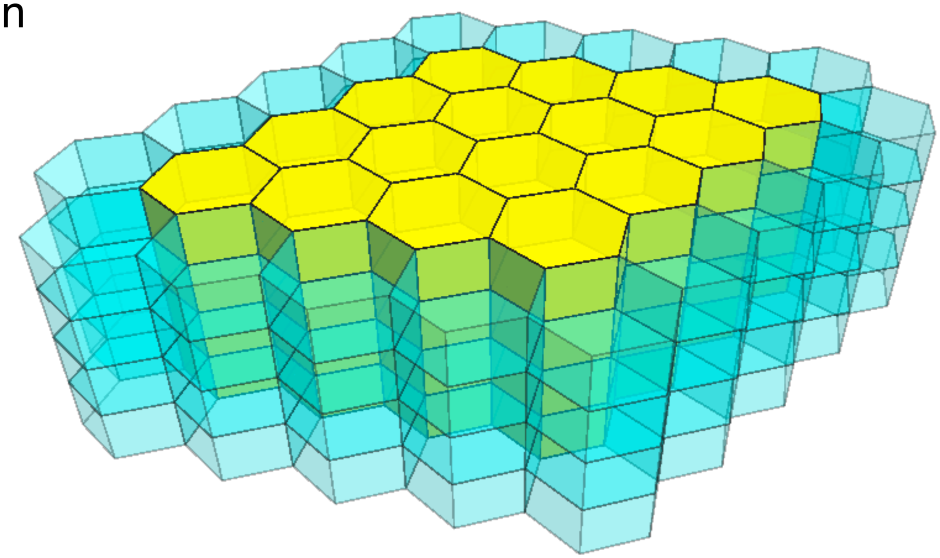


Define parallel efficiency

- Each subdomain block requires information from neighboring subdomains to fill ghost cells.
- We can define a **parallel efficiency** to be

$$\text{parallel efficiency} = \frac{\text{number of local cells}}{\text{number of ghost cells}}$$

- **Larger parallel efficiency is better.**
  More useful work is done per ghost cells.
- For example, **pe** as a function of grid resolution and number of processes



Yellow cells *belong to the local process*

Blue cells *are ghost cells filled from neighboring process*

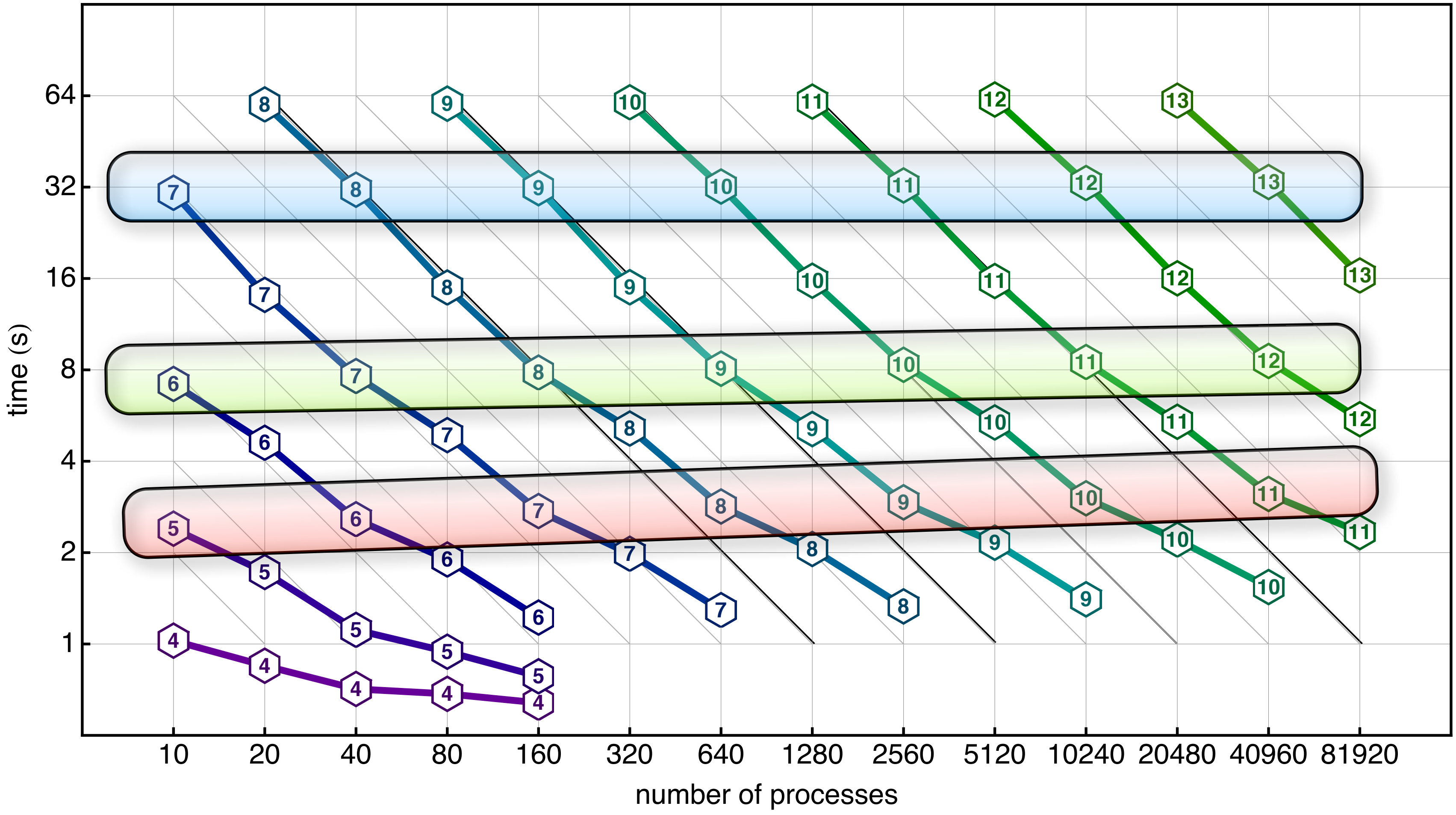
	640	**2560**	**10240**
9 (15.64km)	16	8	4
10 (7.819km)	32	16	8
11 (3.909km)	64	32	16

Parallel Scaling

- What is the relation between parallel efficiency and parallel scalability?

🟦 pe=32
🟩 pe=16
🟥 pe=8

Hopper. 16 V−cycles. 192 layers. grid number is indicated in the hexagon



An MPI/OpenMP hybrid model

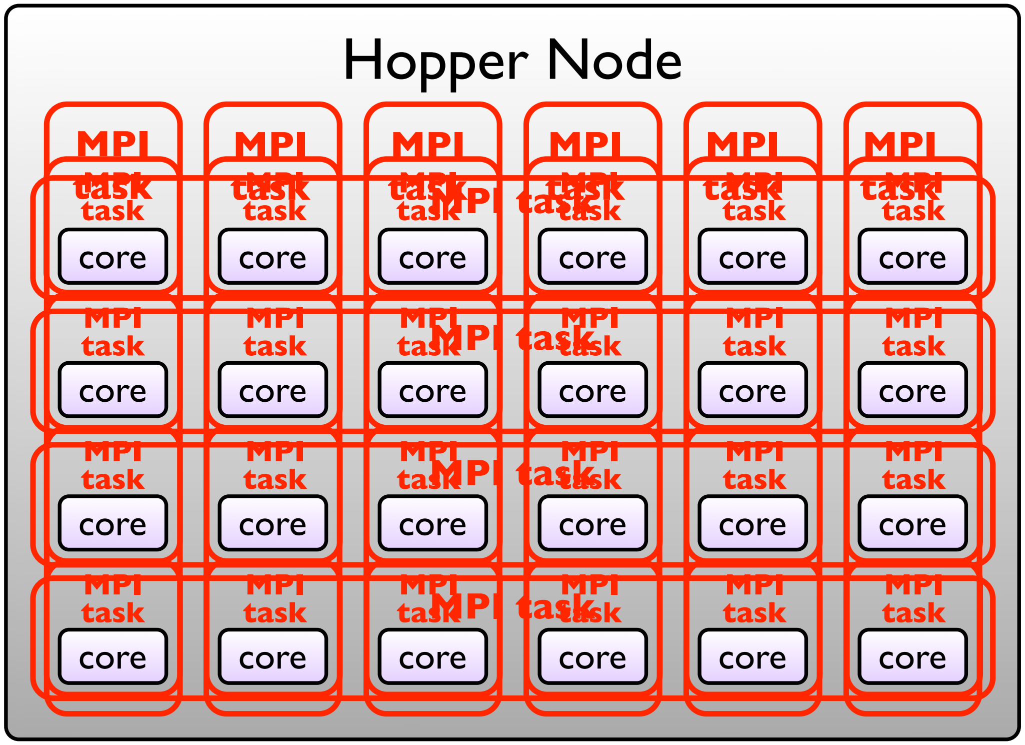
- One possible strategy:

  1) Use MPI parallelism (distributed memory) for the physical domain decomposition such that **pe ≥ 16**

  2) With OpenMP (shared memory) to gain greater parallelism within each MPI task

- Consider 1 node on hopper which has 24 cores

- The same shared memory parallelism ideas apply to GPUs



An MPI/OpenMP hybrid model

- The 2D multigrid is a good place to test the MPI/OpenMP strategy.
- A stack of 2D problems where there are no dependencies across the vertical dimension. The OpenMP parallelization is on the vertical loop.
- Let's look at four experiments:

Grid 9
640 MPI tasks

time = 8.06s

Grid 9
2560 MPI tasks

time = 2.93s

ideal time = ~2s

Grid 9
640 MPI tasks
4 OpenMP Threads
2560 total processes

time = 3.57s

ideal time = ~2s

Grid 9
640 MPI tasks
6 OpenMP Threads
3840 total processes

time=3.26s

ideal time = ~1.33s

- Somewhat disappointing results

A parallel tridiagonal solver using OpenMP

- Code with **dependencies in the vertical direction** will need to be modified to allow parallelism.
- In particular we need to solve tridiagonal systems in the vertical direction

  1) Implicit vertical diffusion processes

  2) In the 3D solver in the unified system
- A tridiagonal system has the form

$$\begin{bmatrix} b_1 & c_1 & & & & & \mathbf{0} \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & a_4 & b_4 & c_4 & & \\ & & & a_5 & b_5 & c_5 & \\ & & & & a_6 & b_6 & c_6 \\ \mathbf{0} & & & & & a_7 & b_7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix}$$

A parallel tridiagonal solver using OpenMP

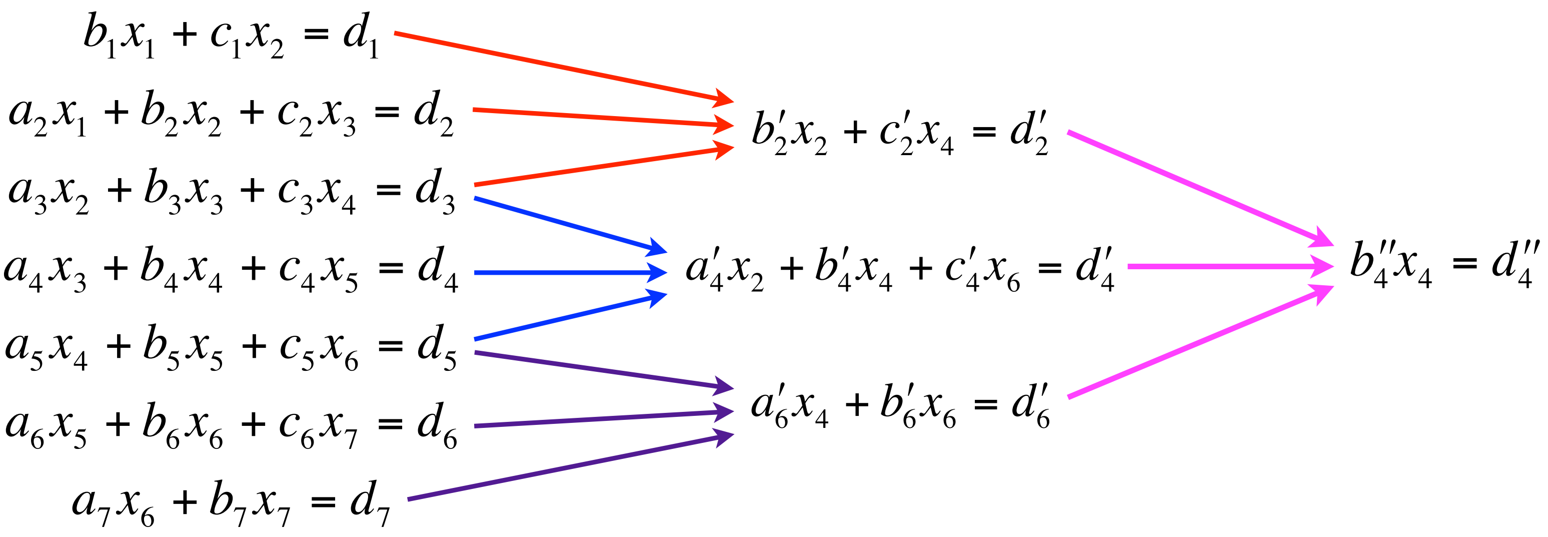
- The conventional algorithm (Numerical Recipes) does not parallelize
- Each result depends on the previous result

```fortran
subroutine solve_tridiag(a,b,c,v,x,n)

        bp(1) = b(1)
        vp(1) = v(1)

        !The first pass (setting coefficients):
firstpass: do i = 2,n
         m = a(i)/bp(i-1)
         bp(i) = b(i) - m*c(i-1)
         vp(i) = v(i) - m*vp(i-1)
        end do firstpass

         x(n) = vp(n)/bp(n)
        !The second pass (back-substition)
backsub:do i = n-1, 1, -1
          x(i) = (vp(i) - c(i)*x(i+1))/bp(i)
        end do backsub

end subroutine solve_tridiag
```

A parallel tridiagonal solver using OpenMP

- The algorithm know as **cyclic reduction** has greater inherent parallelism.

- Consider a 7×7 system of equations:



$$b_1x_1 + c_1x_2 = d_1$$

$$a_2x_1 + b_2x_2 + c_2x_3 = d_2$$

$$a_3x_2 + b_3x_3 + c_3x_4 = d_3$$

$$a_4x_3 + b_4x_4 + c_4x_5 = d_4$$

$$a_5x_4 + b_5x_5 + c_5x_6 = d_5$$

$$a_6x_5 + b_6x_6 + c_6x_7 = d_6$$

$$a_7x_6 + b_7x_7 = d_7$$

$$b_2'x_2 + c_2'x_4 = d_2'$$

$$a_4'x_2 + b_4'x_4 + c_4'x_6 = d_4'$$

$$a_6'x_4 + b_6'x_6 = d_6'$$

$$b_4''x_4 = d_4''$$

- The linear combinations of equations are independent and can proceed in parallel.

A parallel tridiagonal solver using OpenMP

- Let’s look at four experiments:

The **old** algorithm

Gaussian elimination and back substitution

time = 7.4×10^{-3}s

The **new** algorithm

1 OpenMp thread

time = 1.8×10^{-2}s
(2.5 time slower)

The **new** algorithm

4 OpenMp thread

time = 7.8×10^{-3}s
(1.06 time slower)

The **new** algorithm

6 OpenMp thread

time = 6.9×10^{-3}s
(0.93 time slower)

- Again, somewhat disappointing results

progress, conclusions and future work

- I think I have some ideas why the OpenMP is not working too well. On smaller problem sizes, the overhead associated with forking to create new threads is swamping the parallel gains.