Numerical Modelling in FORTRAN day 10

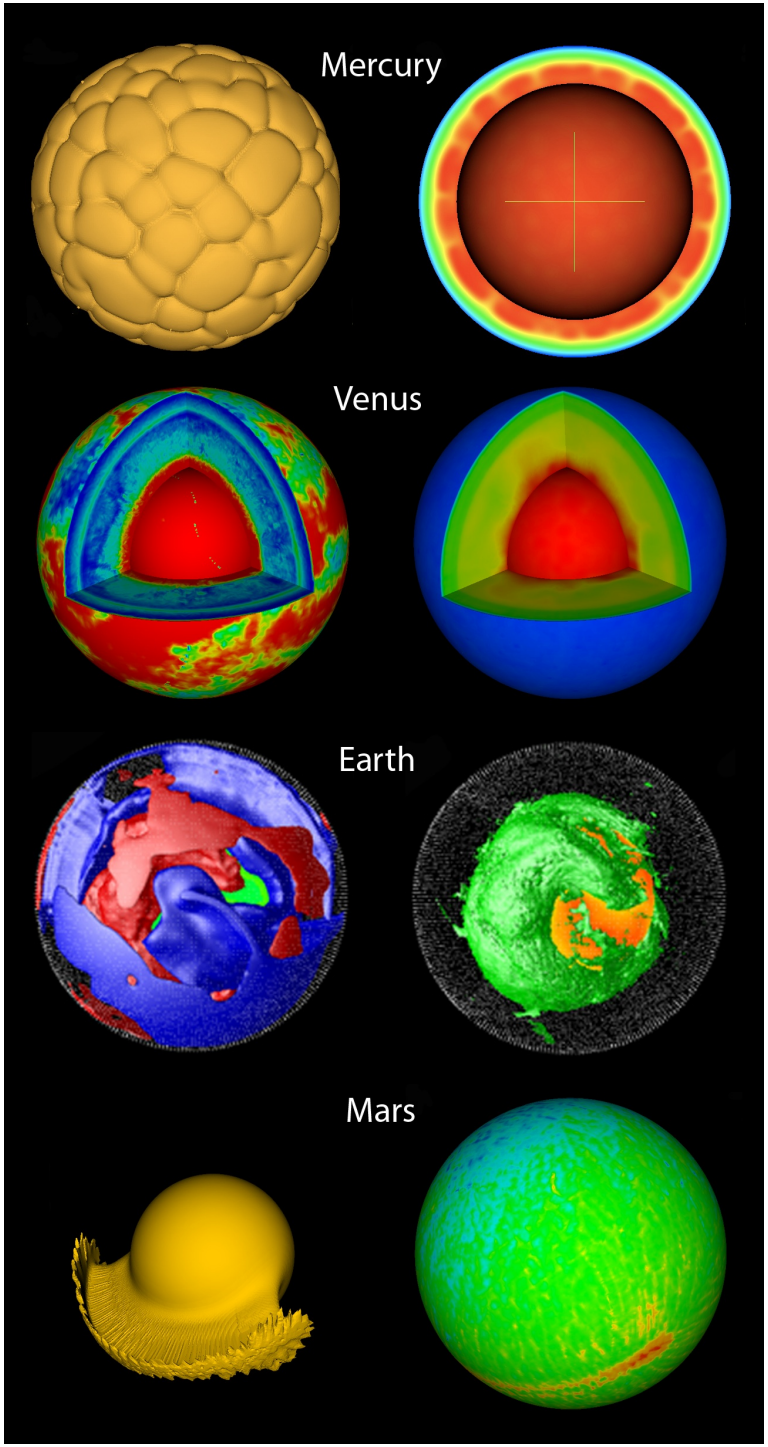
Paul Tackley, 2014

Today's goals

- Introduction to parallel computing (applicable to Fortran or C; examples are in Fortran)
- Continue work on own codes
- Discus projects (for those registered)

Motivation:
To model the Earth, need a huge number of grid points / cells / elements!

- e.g., to fill mantle volume:
  - $(8\ \text{km})^3$ cells -> 1.9 billion cells
  - $(2\ \text{km})^3$ cells -> 123 billion cells



Huge problems => huge computer

www.top500.org

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
8	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301

Huge problems => huge computer

www.top500.org

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
8	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301

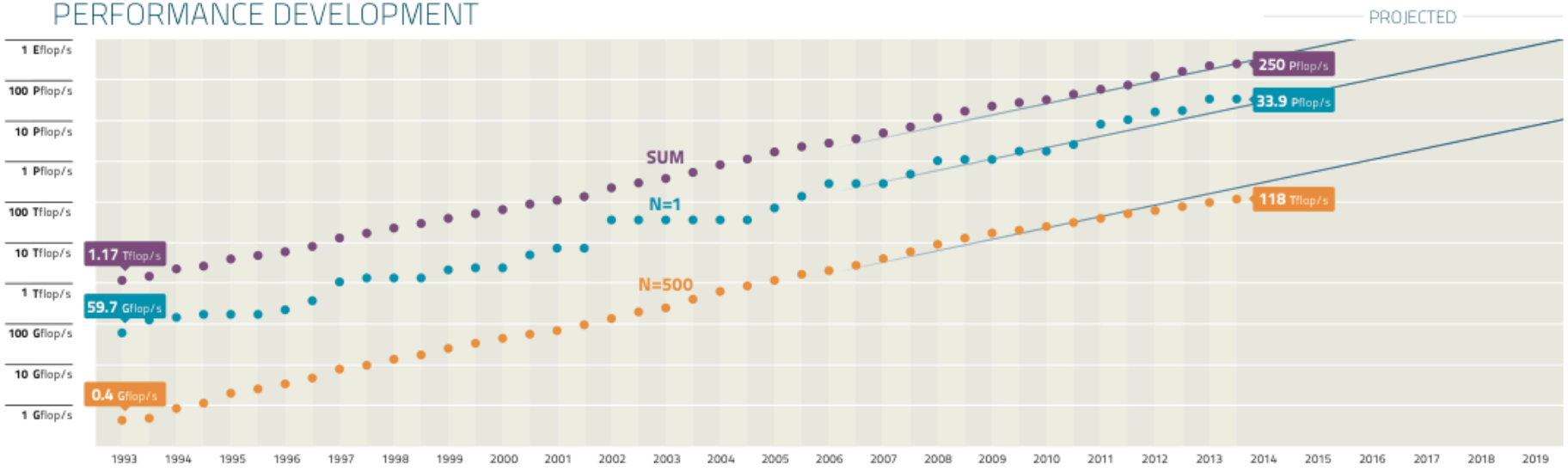
TOP500® NOVEMBER 2013

PRESENTED BY UNIVERSITY OF MANNHEIM · ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE · BERKELEY LAB Lawrence Berkeley National Laboratory

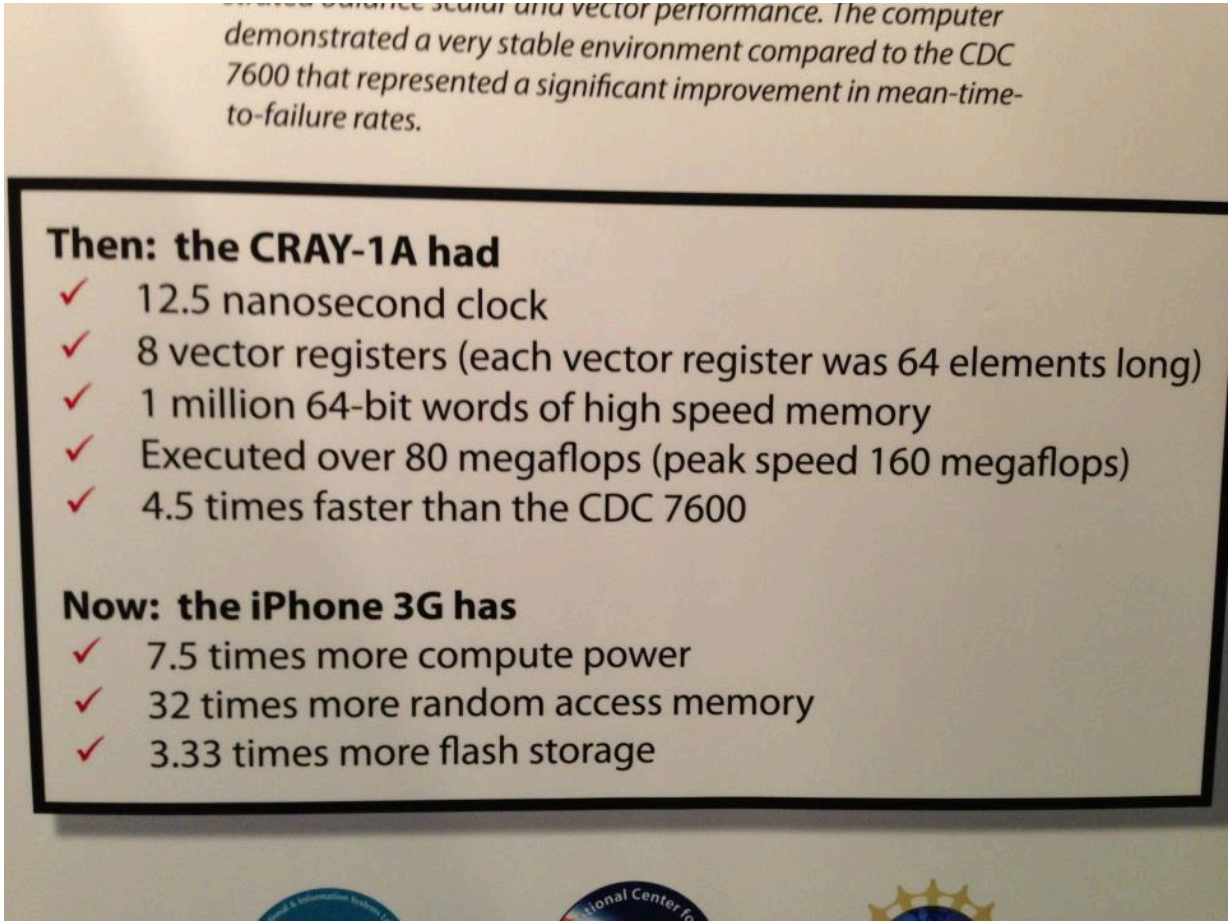
FIND OUT MORE AT **www.top500.org**

	NAME	SPECS	SITE	COUNTRY	CORES	R_{MAX} PFLOP/S	POWER MW
1	**Tianhe-2 (Milkyway-2)**	NUDT, Intel Ivy Bridge (12C, 2.2 GHz) & Xeon Phi (57C, 1.1 GHz), Custom interconnect	NSCC Guangzhou	China	3,120,000	**33.9**	17.8
2	**Titan**	Cray XK7, Operon 6274 (16C 2.2 GHz) + Nvidia Kepler GPU, Custom interconnect	DOE/SC/ORNL	USA	560,640	**17.6**	8.2
3	**Sequoia**	IBM BlueGene/Q, Power BQC (16C 1.60 GHz), Custom interconnect	DOE/NNSA/LLNL	USA	1,572,864	**17.2**	7.9
4	**K computer**	Fujitsu SPARC64 VIIIfx (8C, 2.0GHz), Custom interconnect	RIKEN AICS	Japan	705,024	**10.5**	12.7
5	**Mira**	IBM BlueGene/Q, Power BQC (16C, 1.60 GHz), Custom interconnect	DOE/SC/ANL	USA	786,432	**8.59**	3.95

PERFORMANCE DEVELOPMENT

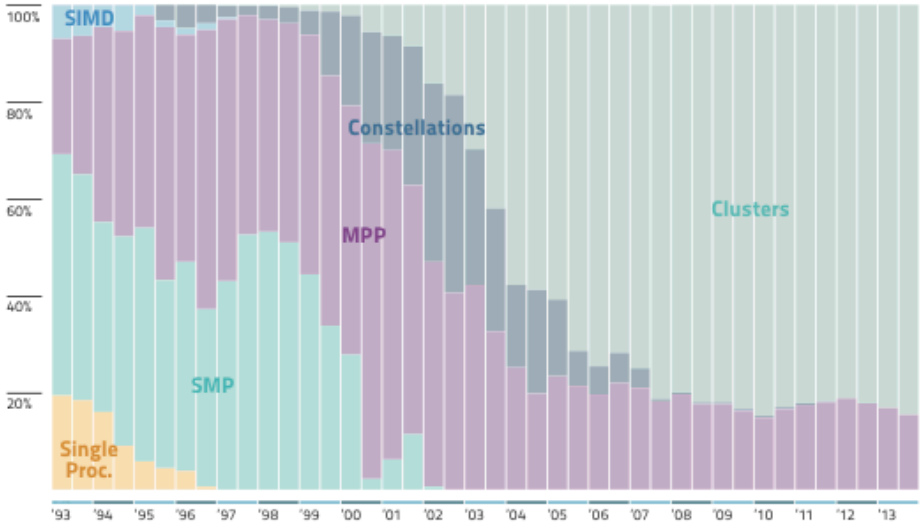


Progress: iPhone > fastest computer in 1976 (cost: $8 million)

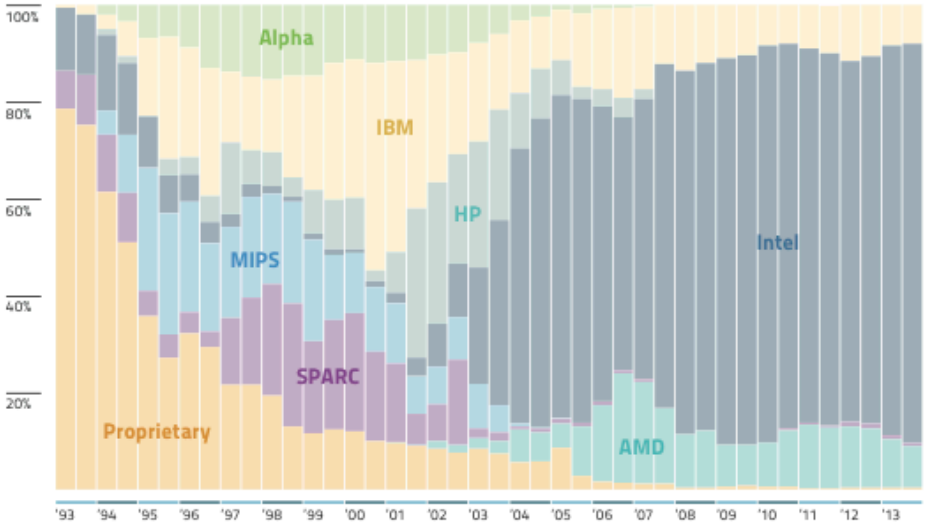


(photo taken at NCAR museum)

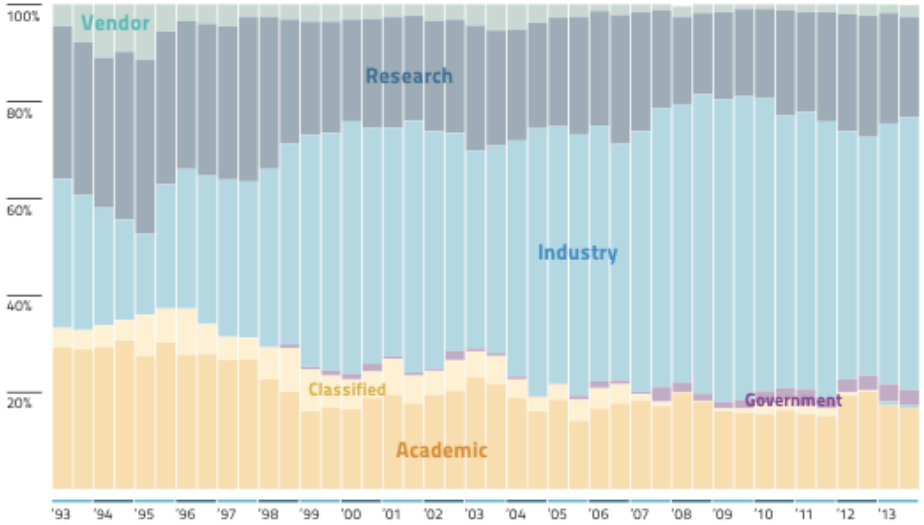
ARCHITECTURES



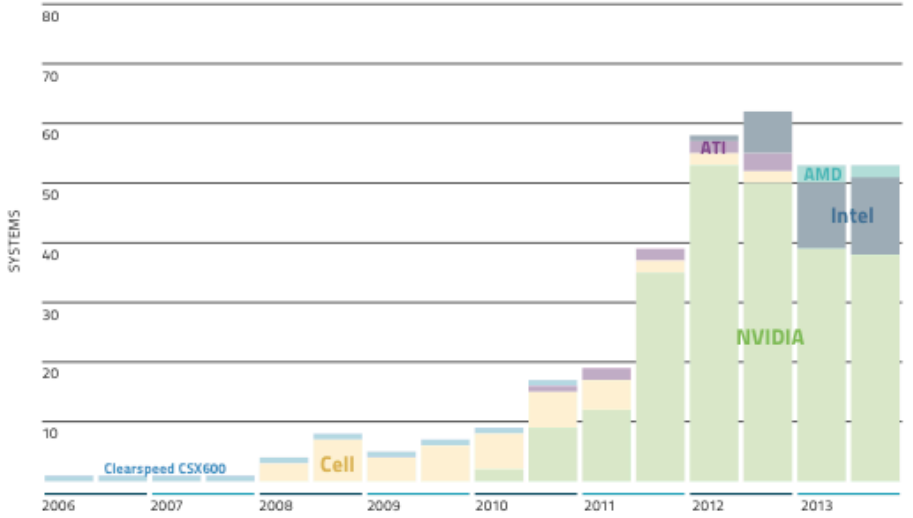
CHIP TECHNOLOGY



INSTALLATION TYPE



ACCELERATORS / CO-PROCESSORS



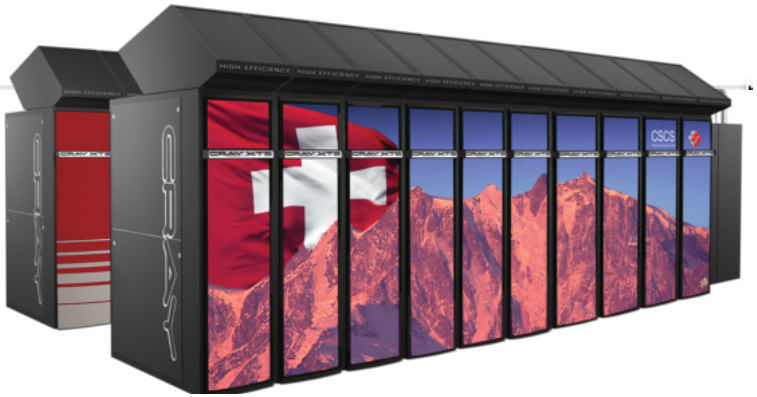
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115984	6271.0	7788.9	2325

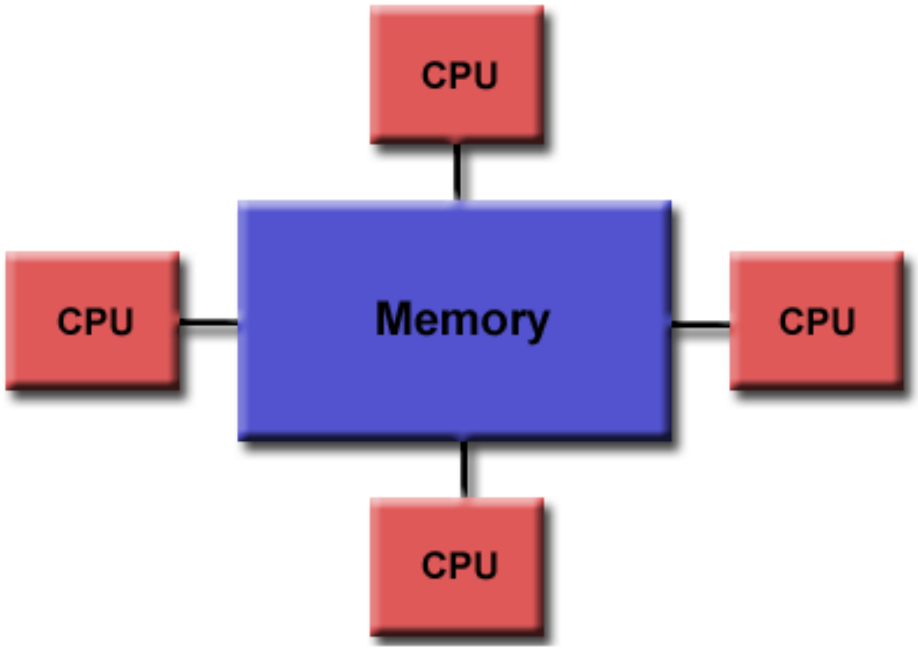
Each node: 8-core Intel + GPU

47	Swiss National Supercomputing Centre (CSCS) Switzerland	**EPFL Blue Brain IV** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	65536	715.6	838.9	329

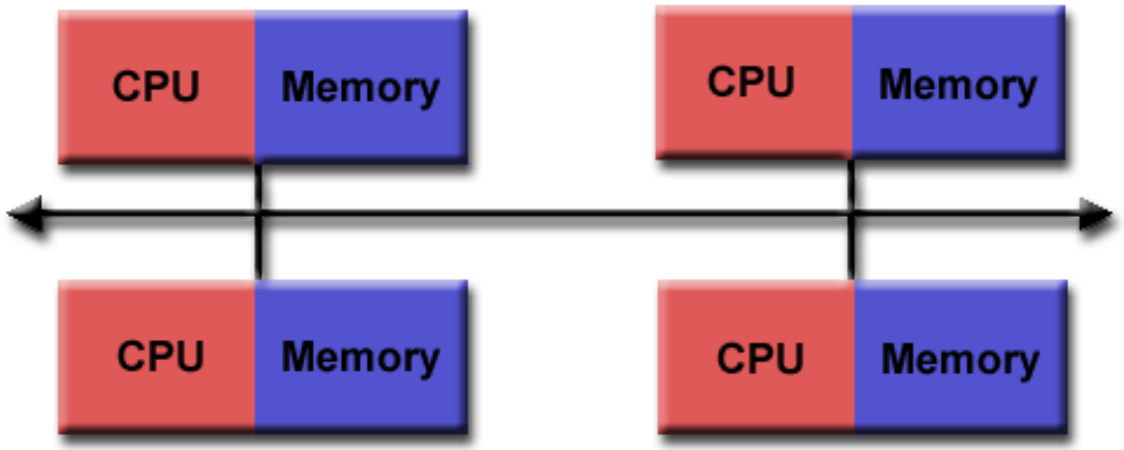
112	Swiss National Supercomputing Centre (CSCS) Switzerland	**Monte Rosa** - Cray XE6, Opteron 6272 16C 2.10 GHz, Cray Gemini interconnect Cray Inc.	47840	316.2	401.9	780

Each node: 2x16-core Opteron





Shared memory: several cpus (or cores) share the same memory. Parallelisation can often be done by the compiler (sometimes with help, e.g., OpenMP instructions in the code)



Distributed memory: each cpu has its own memory. Parallelisation usually requires message-passing, e.g. using MPI (message-passing interface)

A brief history of supercomputers

1983-5
4 CPUs,
Shared
memory

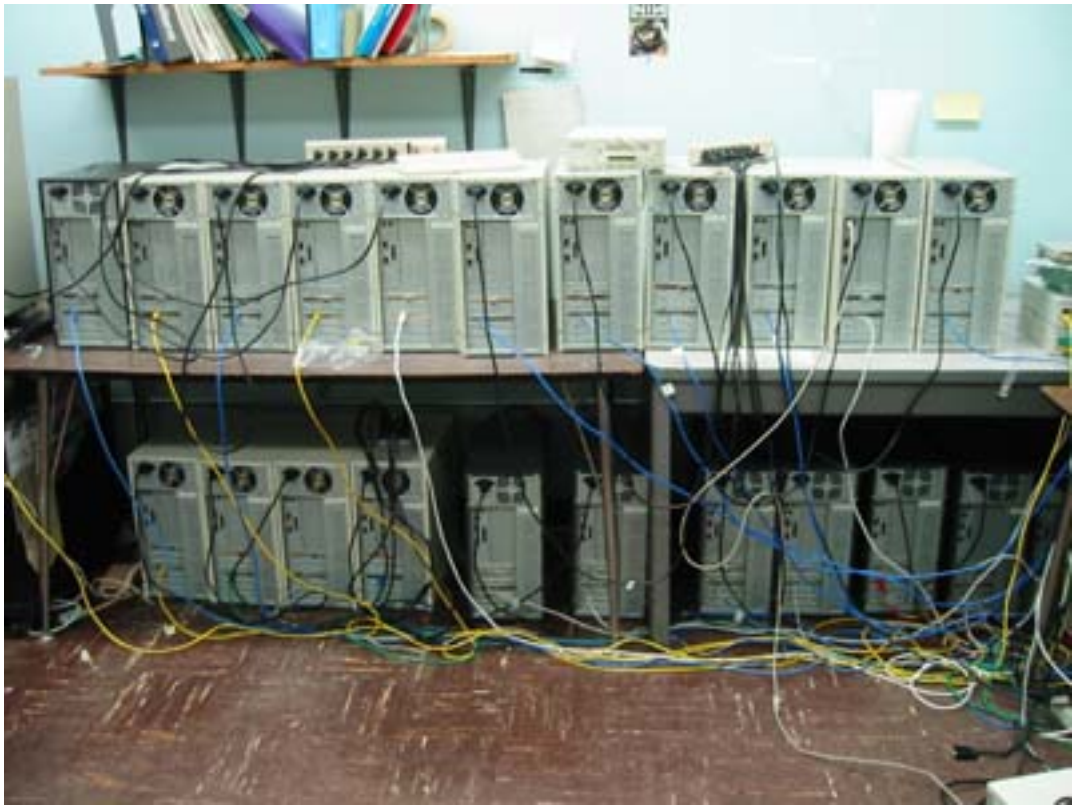
Touchstone Delta delivered to Caltech

1991: 512 CPUs, distributed memory

2010: 224,162 Cores, distributed + shared memory (12 cores per node)

Another possibility: build you own ("Beowulf" cluster)

Using standard PC cases:



or using rack-mounted cases

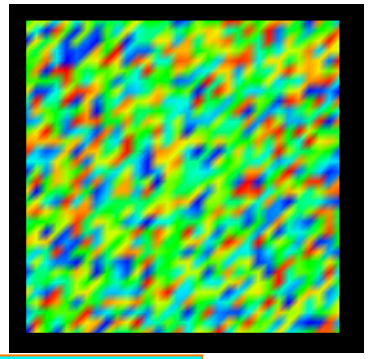
MPI: message-passing interface

- A standard library for communicating between different tasks (cpus)
  - Pass messages (e.g., arrays)
  - Global operations (e.g., sum, maximum)
  - Tasks could be on different cpus/cores of the same node, or on different nodes
- Works with Fortran and C
- Works on everything from a laptop to the largest supercomputers. 2 versions are:
  - http://www.mcs.anl.gov/research/projects/mpich2/
  - http://www.open-mpi.org/

How to parallelise a code: worked example

Example: Scalar Poisson eqn.

$$\nabla^2 u = f$$



Finite-difference approximation:

$$\frac{1}{h^2}\left(u_{i+1\,jk} + u_{i-1\,jk} + u_{ij+1k} + u_{ij-1k} + u_{ijk+1} + u_{ijk+1} - 6u_{i,j}\right) = f_{ij}$$

Use iterative approach=>start with u=0, sweep through grid updating u values according to:

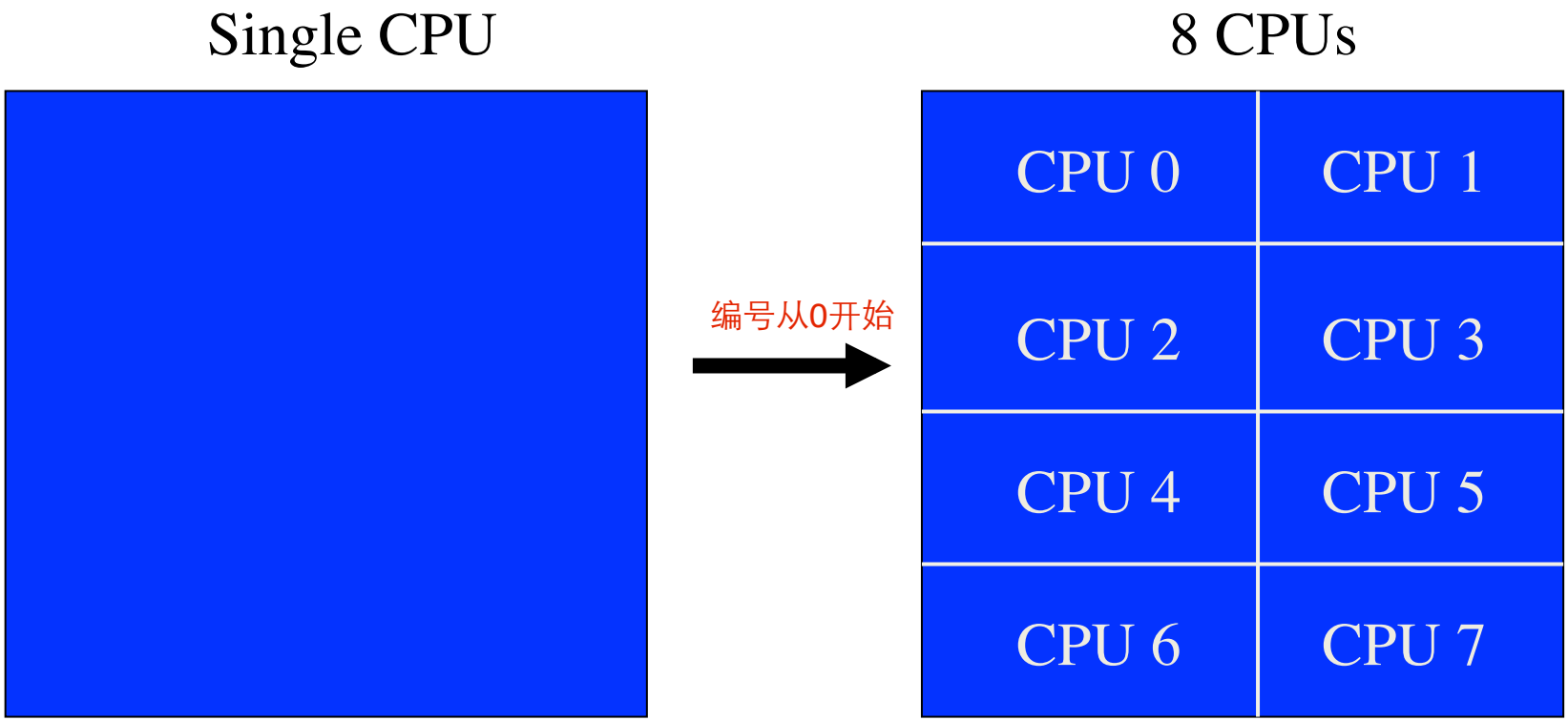
$$\tilde{u}_{ij}^{n+1} = \tilde{u}_{ij}^{n} + \alpha R_{ij}\frac{h^2}{6}$$

Where Rij is the **residue** ("error"):

$$R = \nabla^2 \tilde{u} - f$$

```
program SimplePoisson

  implicit none

  integer:: nx=32,ny=32,nz=32                    ! #grid points
  real   :: convergence_limit=1.e-3, alpha=0.9 ! numerical things
  real    h,resmax
  integer i,j,k, iter
  real,allocatable:: u(:,:,:),f(:,:,:),r(:,:,:)

  ! set up

  allocate (u(0:nx,0:ny,0:nz),f(0:nx,0:ny,0:nz),r(0:nx,0:ny,0:nz))
  u = 0.; f = 0.; r = 0.

  forall (i=1:nx-1, j=1:ny-1, k=1:nz-1) &
       f(i,j,k) = float(i + j + k)/(nx+ny+nz)

  h = 1./nz  ! grid spacing

  ! iterate to convergence

  iter=0; resmax=2*convergence_limit

  do while (resmax>convergence_limit)

     forall (i=1:nx-1, j=1:ny-1, k=1:nz-1)                    &
          r(i,j,k) = ( u(i+1,j,k) + u(i-1,j,k)                &
                     + u(i,j+1,k) + u(i,j-1,k)                &
                     + u(i,j,k+1) + u(i,j,k-1)                &
                   - 6*u(i,j,k))/h**2 - f(i,j,k)

     u = u + alpha * h**2/6 * r

     iter = iter + 1; resmax = maxval(abs(r))
     print*,iter,resmax

  end do

end program SimplePoisson
```

Parallelisation: domain decomposition



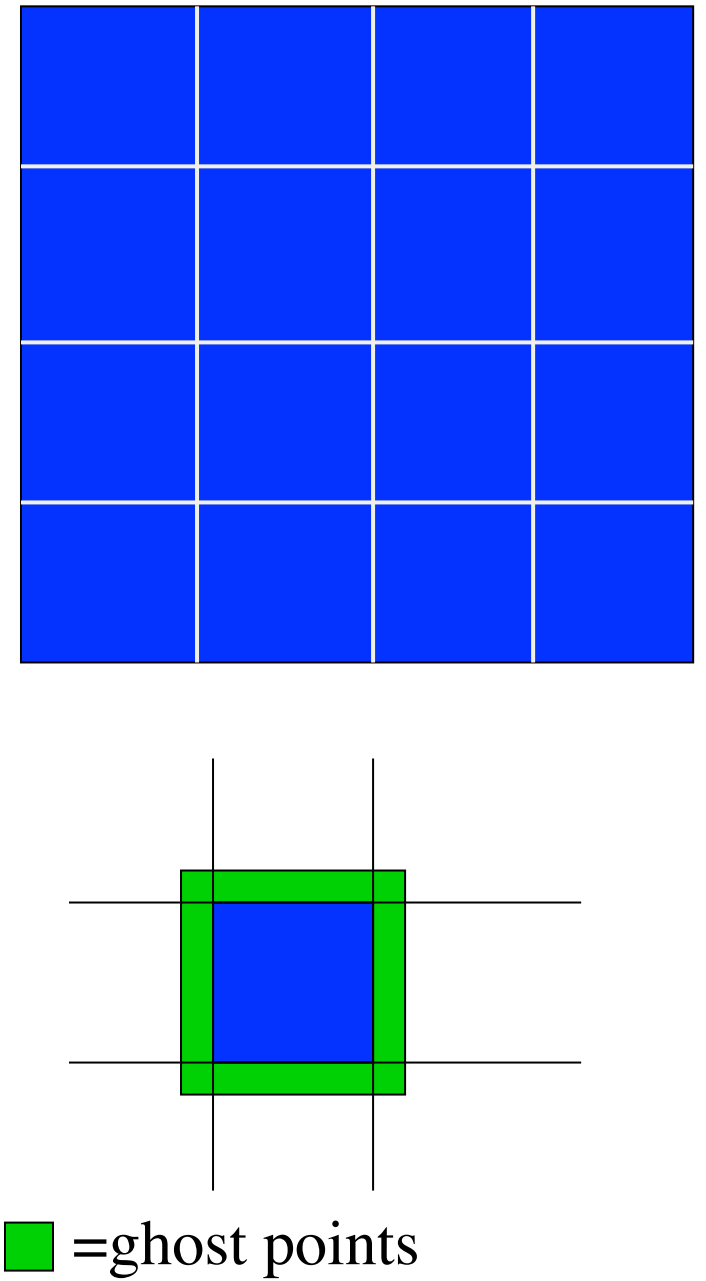
Each CPU will do the same operations but on different parts of the domain

You need to build parallelization into the code using MPI

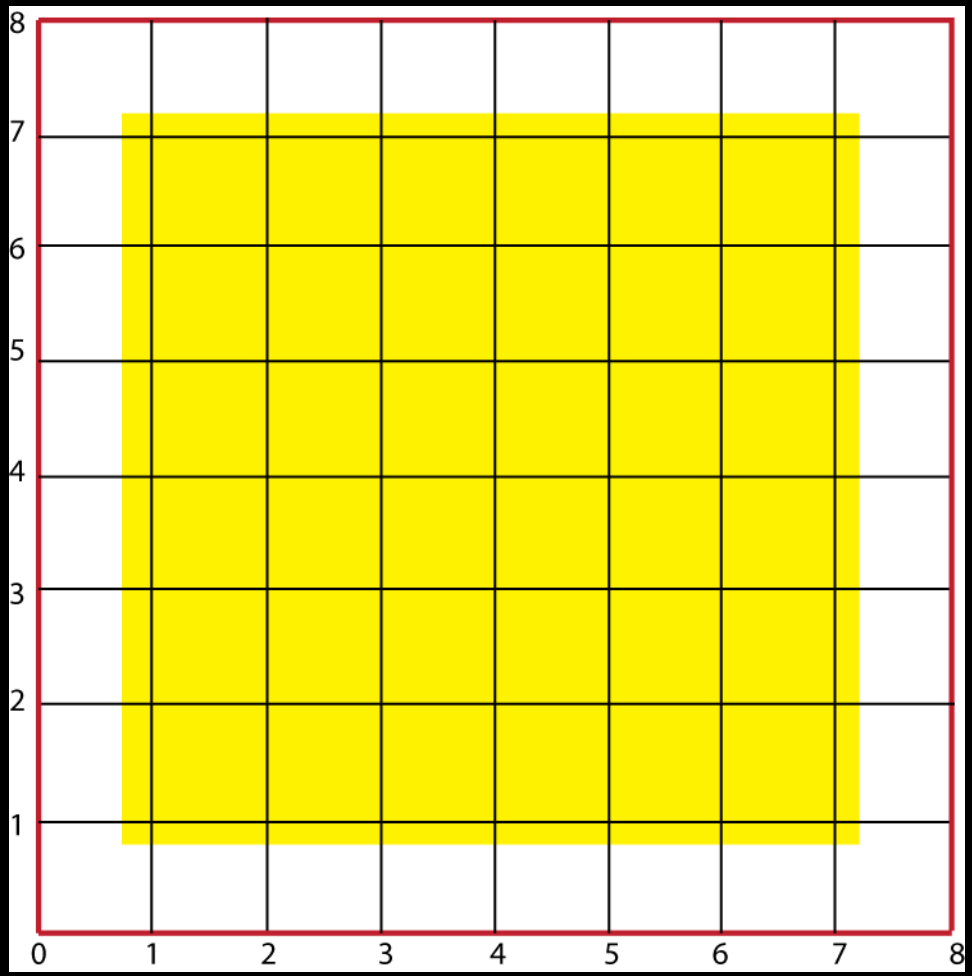
- Any scalar code will run on multiple CPUs, but will produce the same result on each CPU.
- Code must first setup local grid in relation to global grid, then handle communication
- Only a few MPI calls needed:
  - Init. (MPI_init,MPI_com_size,MPI_com_rank) initialisation
  - Global combinations (MPI_allreduce)
  - CPU-CPU communication (MPI_send,MPI_recv…)

Boundaries

- When updating points at edge of subdomain, need values on neighboring subdomains
- Hold copies of these locally using “ghost points”
- This minimizes #of messages, because they can be updated all at once instead of individually



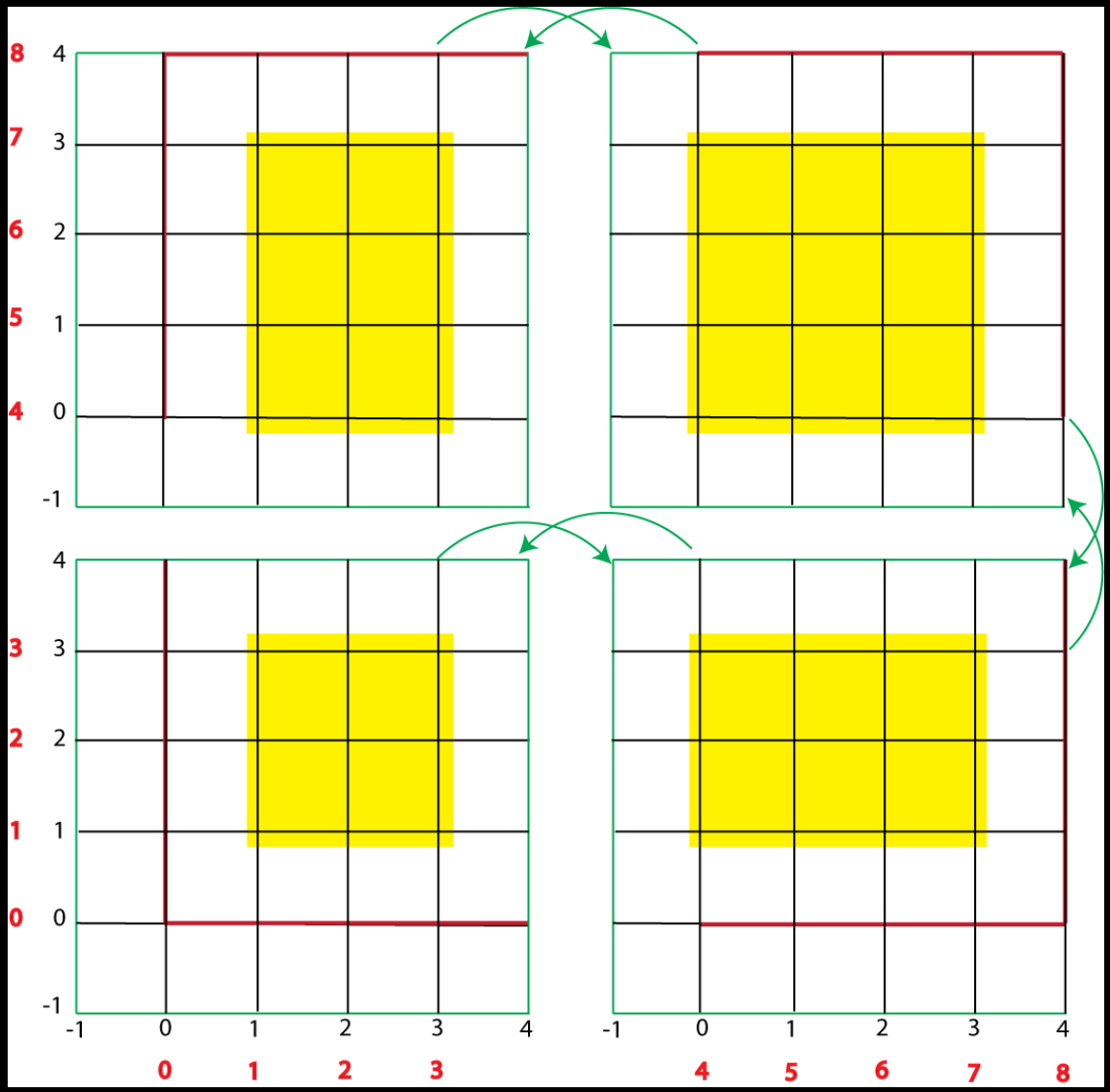
Scalar Grid



Red=boundary points (=0)

Yellow=iterated/solved (1…n-1)

Parallel grids



Red=ext. boundaries

Green=int. boundaries

Yellow=iterated/solved

复制了临近的点

First things the code has to do:

- Call MPI_init(ierr)
- Find #CPUs using MPI_com_size
- Find which CPU it is, using MPI_com_rank (returns a number from 0…#CPUs-1)
- Calculate which part of the global grid it is dealing with, and which other CPUs are handling neighboring subdomains.

Example: “Hello world” program

```
program HelloMPI

  use mpi

  implicit none

  integer ierr, ncpus, mycpu

  call MPI_init(ierr)
  call MPI_comm_size(MPI_comm_world,ncpus,ierr)
  call MPI_comm_rank(MPI_comm_world,mycpu,ierr)

  print*,'Hello from cpu number :',mycpu,' of ',ncpus

  call MPI_finalize(ierr)

end program HelloMPI
```

an integer, always the last parameter

close the library

To compile it, you need “mpif90
Hellow .f90”, also use ‘mpirun -n 4 a.out’
to know the number of MPI running.

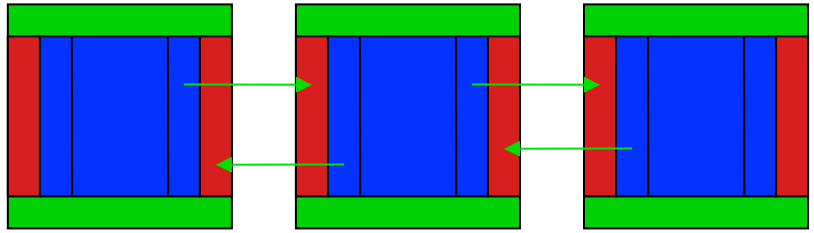
如果你的电脑有两个核，则不要选择大于2的核心数令其运算

Moving forward

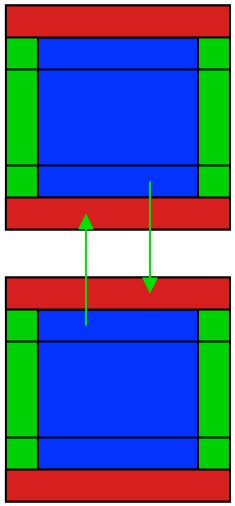
- Update values in subdomain using ‘ghost points’ as boundary condition, i.e.,
  - Timestep (explicit), or
  - Iteration (implicit)
- Update ghost points by communicating with other CPUs
- Works well for explicit or iterative approaches

Boundary communication

Step 1: x-faces



Step 2: y-faces (including corner values from step 1)



[Step 3: z-faces (including corner values from steps 1 & 2)]

Doing the 3 directions sequentially avoids the need for additional messages to do edges & corners (=>in 3D, 6 messages instead of 26)

```
program SimplePoissonMPI

  use mpi

  implicit none

  integer:: nx=32,ny=32,nz=32                  ! #grid points in global domain
  real   :: convergence_limit=1.e-3, alpha=0.9 ! numerical parameters
  real    h,resmax
  integer i,j,k, iter
  real,allocatable:: u(:,:,:),f(:,:,:),r(:,:,:)

  integer ncpus,mycpu           ! #cpus and number of local cpu
  integer nxl,nyl,nzl           ! #grid points in local subdomain
  integer xmin,ymin,zmin        !  min. coordinate (1 if external bndry, 0 if internal)
  integer:: ncx=1,ncy=1,ncz=1   ! #cpus in each direction
  integer myx,myy,myz           !  local subdomain coordinates
  integer cpu_xm,cpu_xp,cpu_ym,cpu_yp,cpu_zm,cpu_zp ! cpus holding adjacent subdomains
  integer npx,npy,npz           ! #points along each side (to communicate)
  integer ierr                  !  needed for MPI routines in Fortran

  ! set up

  call set_up_parallelisation()

  allocate (u(-1:nxl,-1:nyl,-1:nzl),f(-1:nxl,-1:nyl,-1:nzl),r(-1:nxl,-1:nyl,-1:nzl))
  u = 0.; f = 0.; r = 0.

  forall (i=xmin:nxl-1, j=ymin:nyl-1, k=zmin:nzl-1)                    &
       f(i,j,k) = float(i+myx*nxl + j+myy*nyl + k+myz*nzl)/(nx+ny+nz)

  h = 1./nz  ! grid spacing

  ! iterate to convergence

  iter=0; resmax=2*convergence_limit

  do while (resmax>convergence_limit)

     forall (i=xmin:nxl-1, j=ymin:nyl-1, k=zmin:nzl-1)          &
          r(i,j,k) = (   u(i+1,j,k) + u(i-1,j,k)                &
          +              u(i,j+1,k) + u(i,j-1,k)                &
          +              u(i,j,k+1) + u(i,j,k-1)                &
          -           6*u(i,j,k))/h**2 - f(i,j,k)

     u = u + alpha * h**2/6 * r

     call update_sides()

     iter = iter + 1; resmax = maxval(abs(r)); call simple_globalmax(resmax)
     if (mycpu==0) print*,iter,resmax

  end do

  call MPI_finalize (ierr)
```

```
program SimplePoissonMPI

  use mpi

  implicit none

  integer:: nx=32,ny=32,nz=32                  ! #grid points in global domain
  real   :: convergence_limit=1.e-3, alpha=0.9 ! numerical parameters
  real    h,resmax
  integer i,j,k, iter
  real,allocatable:: u(:,:,:),f(:,:,:),r(:,:,:)

  integer ncpus,mycpu           ! #cpus and number of local cpu
  integer nxl,nyl,nzl           ! #grid points in local subdomain
  integer xmin,ymin,zmin        !  min. coordinate (1 if external bndry, 0 if internal)
  integer:: ncx=1,ncy=1,ncz=1   ! #cpus in each direction
  integer myx,myy,myz           !  local subdomain coordinates
  integer cpu_xm,cpu_xp,cpu_ym,cpu_yp,cpu_zm,cpu_zp ! cpus holding adjacent subdomains
  integer npx,npy,npz           ! #points along each side (to communicate)
  integer ierr                  !  needed for MPI routines in Fortran

  ! set up

  call set_up_parallelisation()

  allocate (u(-1:nxl,-1:nyl,-1:nzl),f(-1:nxl,-1:nyl,-1:nzl),r(-1:nxl,-1:nyl,-1:nzl))
  u = 0.; f = 0.; r = 0.

  forall (i=xmin:nxl-1, j=ymin:nyl-1, k=zmin:nzl-1)                    &
       f(i,j,k) = float(i+myx*nxl + j+myy*nyl + k+myz*nzl)/(nx+ny+nz)

  h = 1./nz  ! grid spacing

  ! iterate to convergence

  iter=0; resmax=2*convergence_limit

  do while (resmax>convergence_limit)

     forall (i=xmin:nxl-1, j=ymin:nyl-1, k=zmin:nzl-1)          &
          r(i,j,k) = (   u(i+1,j,k) + u(i-1,j,k)                &
          +              u(i,j+1,k) + u(i,j-1,k)                &
          +              u(i,j,k+1) + u(i,j,k-1)                &
```

```
integer ncpus,mycpu           ! #cpus and number of local cpu
integer nxl,nyl,nzl           ! #grid points in local subdomain
integer xmin,ymin,zmin        !  min. coordinate (1 if external bndry, 0 if internal)
integer:: ncx=1,ncy=1,ncz=1   ! #cpus in each direction
integer myx,myy,myz           !  local subdomain coordinates
integer cpu_xm,cpu_xp,cpu_ym,cpu_yp,cpu_zm,cpu_zp ! cpus holding adjacent subdomains
integer npx,npy,npz           ! #points along each side (to communicate)
integer ierr                  !  needed for MPI routines in Fortran

! set up

call set_up_parallelisation()

allocate (u(-1:nxl,-1:nyl,-1:nzl),f(-1:nxl,-1:nyl,-1:nzl),r(-1:nxl,-1:nyl,-1:nzl))
u = 0.; f = 0.; r = 0.

forall (i=xmin:nxl-1, j=ymin:nyl-1, k=zmin:nzl-1)                    &
     f(i,j,k) = float(i+myx*nxl + j+myy*nyl + k+myz*nzl)/(nx+ny+nz)

h = 1./nz  ! grid spacing

! iterate to convergence

iter=0; resmax=2*convergence_limit

do while (resmax>convergence_limit)

   forall (i=xmin:nxl-1, j=ymin:nyl-1, k=zmin:nzl-1)          &
        r(i,j,k) = (   u(i+1,j,k) + u(i-1,j,k)                &
        +              u(i,j+1,k) + u(i,j-1,k)                &
        +              u(i,j,k+1) + u(i,j,k-1)                &
        -           6*u(i,j,k))/h**2 - f(i,j,k)

   u = u + alpha * h**2/6 * r

   call update_sides()

   iter = iter + 1; resmax = maxval(abs(r)); call simple_globalmax(resmax)
   if (mycpu==0) print*,iter,resmax

end do

call MPI_finalize (ierr)
```

Main changes

- Parallelisation hidden in set_up_parallelisation and update_sides
- Many new variables to store parallelisation information
- Loop limits depend on whether global domain boundary or local subdomain

```
contains

  subroutine set_up_parallelisation()
    integer n,nmax
    call MPI_init(ierr)
    call MPI_comm_size(MPI_comm_world,ncpus,ierr)
    call MPI_comm_rank(MPI_comm_world,mycpu,ierr)

    n=1; nxl=nx; nyl=ny; nzl=nz  ! figure out #cpus in each direction
    do while (n<ncpus)           !  such as to keep subdomains ~cubic
       nmax = max(nxl,nyl,nzl)
       if(nxl==nmax) then
          ncx = ncx*2; nxl = nxl/2
       else if (nyl==nmax) then
          ncy = ncy*2; nyl = nyl/2
       else
          ncz = ncz*2; nzl = nzl/2
       end if
       n = n*2
    end do
    if(mycpu==0) print*,'#cpus in each direction:',ncx,ncy,ncz

    myz = mycpu / (ncx*ncy)             ! position of local subdomain
    myy = mod(mycpu,ncx*ncy)/ncx
    myx = mod(mycpu,ncx)

    cpu_xm = mycpu-1      ; cpu_xp = mycpu+1  ! cpus holding adjacent subdomains
    cpu_ym = mycpu-ncx    ; cpu_yp = mycpu+ncx
    cpu_zm = mycpu-ncx*ncy; cpu_zp = mycpu+ncx*ncy

    xmin=1; if (myx>0) xmin=0  ! lowest coordinate to solve for
    ymin=1; if (myy>0) ymin=0  !  0 if internal boundary
    zmin=1; if (myz>0) zmin=0  !  1 if external boundary

    npx=(nyl+2)*(nzl+2)    ! #points on each side to communicate
    npy=(nxl+2)*(nzl+2)
    npz=(nxl+2)*(nyl+2)
  end subroutine set_up_parallelisation
```

Simplest communication

```
  subroutine update_sides()
    if (myx>0     ) call simple_sendrecv(u(   0 , :    , :   ),npx,cpu_xm,u(-1  , : , : )) ! x
    if (myx<ncx-1) call simple_sendrecv(u(nxl-1, :    , :   ),npx,cpu_xp,u(nxl , : , : ))
    if (myy>0     ) call simple_sendrecv(u( :    ,    0, :   ),npy,cpu_ym,u( : ,-1  , : )) ! y
    if (myy<ncy-1) call simple_sendrecv(u( :    ,nyl-1, :   ),npy,cpu_yp,u( : ,nyl , : ))
    if (myz>0     ) call simple_sendrecv(u( :    , :    ,   0 ),npz,cpu_zm,u( : , : , -1)) ! z
    if (myz<ncz-1) call simple_sendrecv(u( :    , :    ,nzl-1),npz,cpu_zp,u( : , : ,nzl))
  end subroutine update_sides

  subroutine simple_sendrecv (sendbuf,length,othercpu,recvbuf)
    integer,intent(in):: length,othercpu
    real,dimension(length):: sendbuf,recvbuf
    integer ierr
    call MPI_sendrecv (sendbuf,length,MPI_real,othercpu,mycpu, &
         recvbuf,length,MPI_real,othercpu,othercpu,MPI_comm_world,ierr)
  end subroutine simple_sendrecv

  subroutine simple_globalmax (buf)
    real buf,work
    call MPI_Allreduce(buf,work,1,MPI_real,MPI_max,MPI_comm_world,ierr)
    buf = work
  end subroutine simple_globalmax

end program simplePoissonMPI
```

Not optimal – uses blocking send/receive

Better: using non-blocking (isend/irecv)

```
  subroutine update_sides()
    integer m(2),ierr
    call sides1d(npx,  &
         myx>0    ,u(0    ,:,:),u(-1 ,:,:),cpu_xm, &
         myx<ncx-1,u(nx1-1,:,:),u(nx1,:,:),cpu_xp )
    call sides1d(npy,  &
         myy>0    ,u(:,    0,:),u(:, -1,:),cpu_ym,  &
         myy<ncy-1,u(:,ny1-1,:),u(:,ny1,:),cpu_yp )
    call sides1d(npz,  &
         myz>0    ,u(:,:,    0),u(:,:, -1),cpu_zm,  &
         myz<ncz-1,u(:,:,nz1-1),u(:,:,nz1),cpu_zp )
  end subroutine update_sides

  subroutine sides1d( length, &
       do_mside,sendbuf_m,recvbuf_m,cpu_m, &
       do_pside,sendbuf_p,recvbuf_p,cpu_p )
    logical,intent(in):: do_mside,do_pside
    integer,intent(in):: cpu_m,cpu_p,length
    real   ,intent(in):: sendbuf_m(length),sendbuf_p(length)
    real  ,intent(out):: recvbuf_m(length),recvbuf_p(length)
    integer m(2),ierr
    if (do_mside) then
       call MPI_isend (sendbuf_m,length,MPI_real,cpu_m,1,MPI_comm_world,m(1),ierr)
       call MPI_irecv (recvbuf_m,length,MPI_real,cpu_m,1,MPI_comm_world,m(2),ierr)
    end if
    if (do_pside) then
       call MPI_send (sendbuf_p,length,MPI_real,cpu_p,1,MPI_comm_world,ierr)
       call MPI_recv (recvbuf_p,length,MPI_real,cpu_p,1,MPI_comm_world,MPI_status_ignore,ierr)
    end if
    if (do_mside) call MPI_waitall (2,m,MPI_status_ignore,ierr)
  end subroutine sides1d

  subroutine simple_globalmax (buf)
    real buf,work
    call MPI_Allreduce(buf,work,1,MPI_real,MPI_max,MPI_comm_world,ierr)
    buf = work
  end subroutine simple_globalmax

end program simplePoissonMPI
```

Performance: theoretical analysis

How much time is spent communicating?

- Computation time $\propto$ volume (N_x^3)
- Communication time $\propto$ surf. area (N_x^2)
- =>Communication/Computation $\propto 1/N_x$
- =>Have as many points/cpu as possible!

Is it better to split 1D, 2D or 3D?

- E.g., 256x256x256 points on 64 CPUs
- 1D split: 256x256x4 points/cpu
  - Area=2x(256x256)=131,072
- 2D split: 256x32x32 points/cpu
  - Area=4x(256x32)=32,768
- 3D split:64x64x64 points/cpu
  - Area=6x(64x64)=24,576
- =>3D best but more messages needed

Model code performance

(Time per step or iteration)

Computation : $t=aN^3$

Communication : $t=nL+bN^2/B$

(L=Latency, B=bandwidth)

TOTAL : $t= aN^3 +nL+bN^2/B$

Example: Scalar Poisson equation

$$\nabla^2 u = f$$

$$t = aN^3 + nL + bN^2/B$$

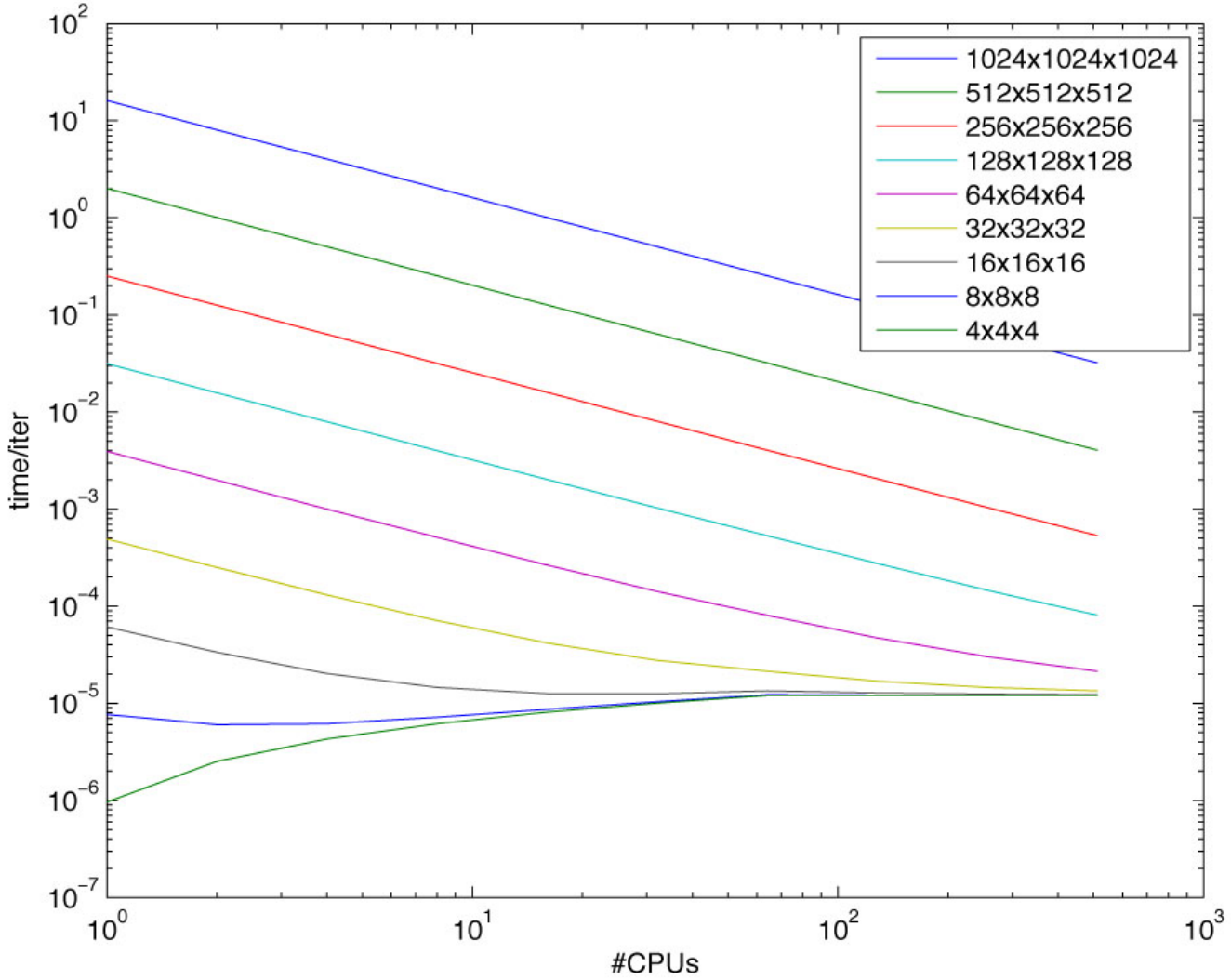
Assume 15 operations/point/iteration & 1 Gflop performance
Þa=15/1e9=1.5e-8

If 3D decomposition, n=6, b=6*4 (single precision)

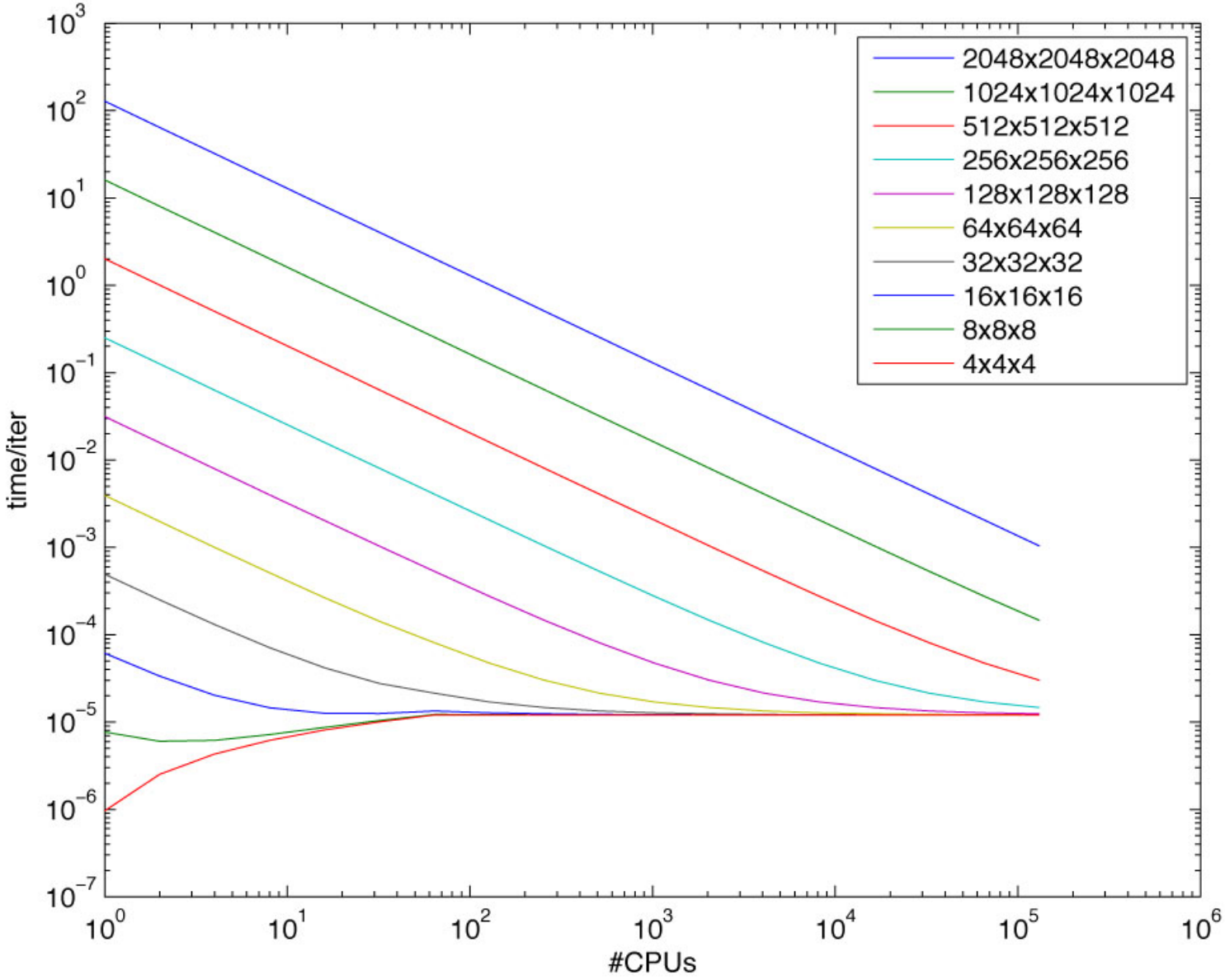
Gigabit ethernet: L=40e-6 s, B=100 MB/s
Quadrics: L=2e-6 s, B=875 MB/s

Time/iteration vs. #cpus

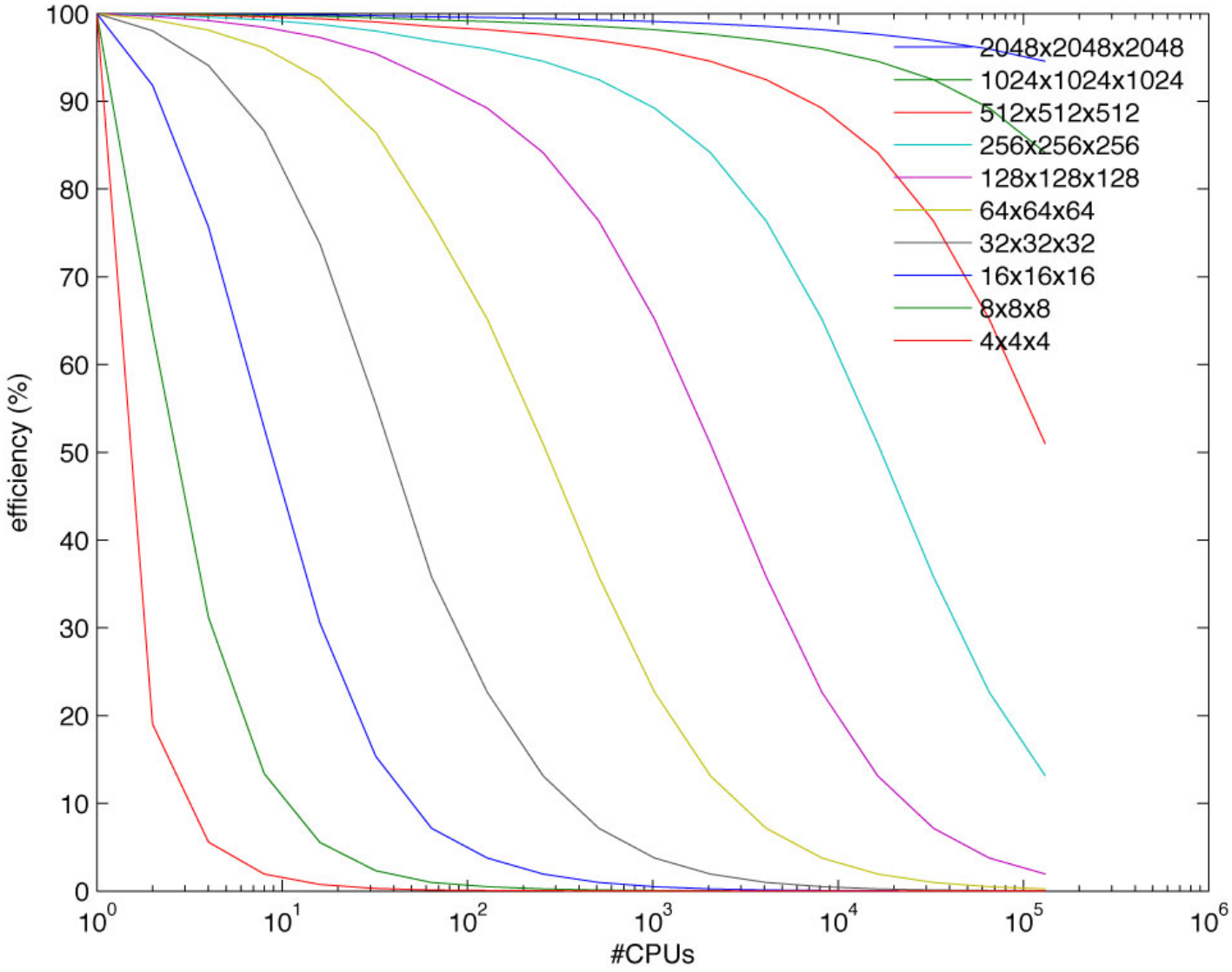
Quadrics, Gonzales-size cluster



Up to 2e5 CPUs (Quadrics communication)



Efficiency


efficiency (%)
#CPUs
2048x2048x2048
1024x1024x1024
512x512x512
256x256x256
128x128x128
64x64x64
32x32x32
16x16x16
8x8x8
4x4x4

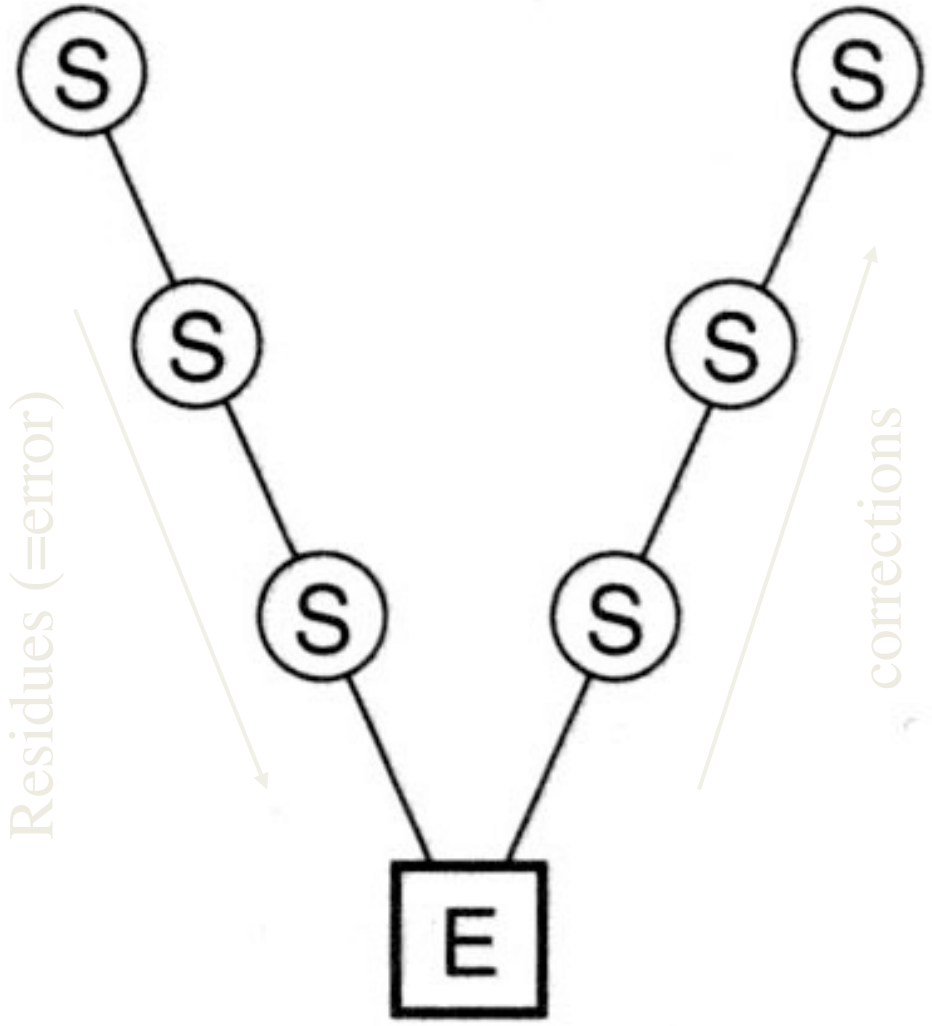
Now multigrid V cycles

Smooth 32x32x32

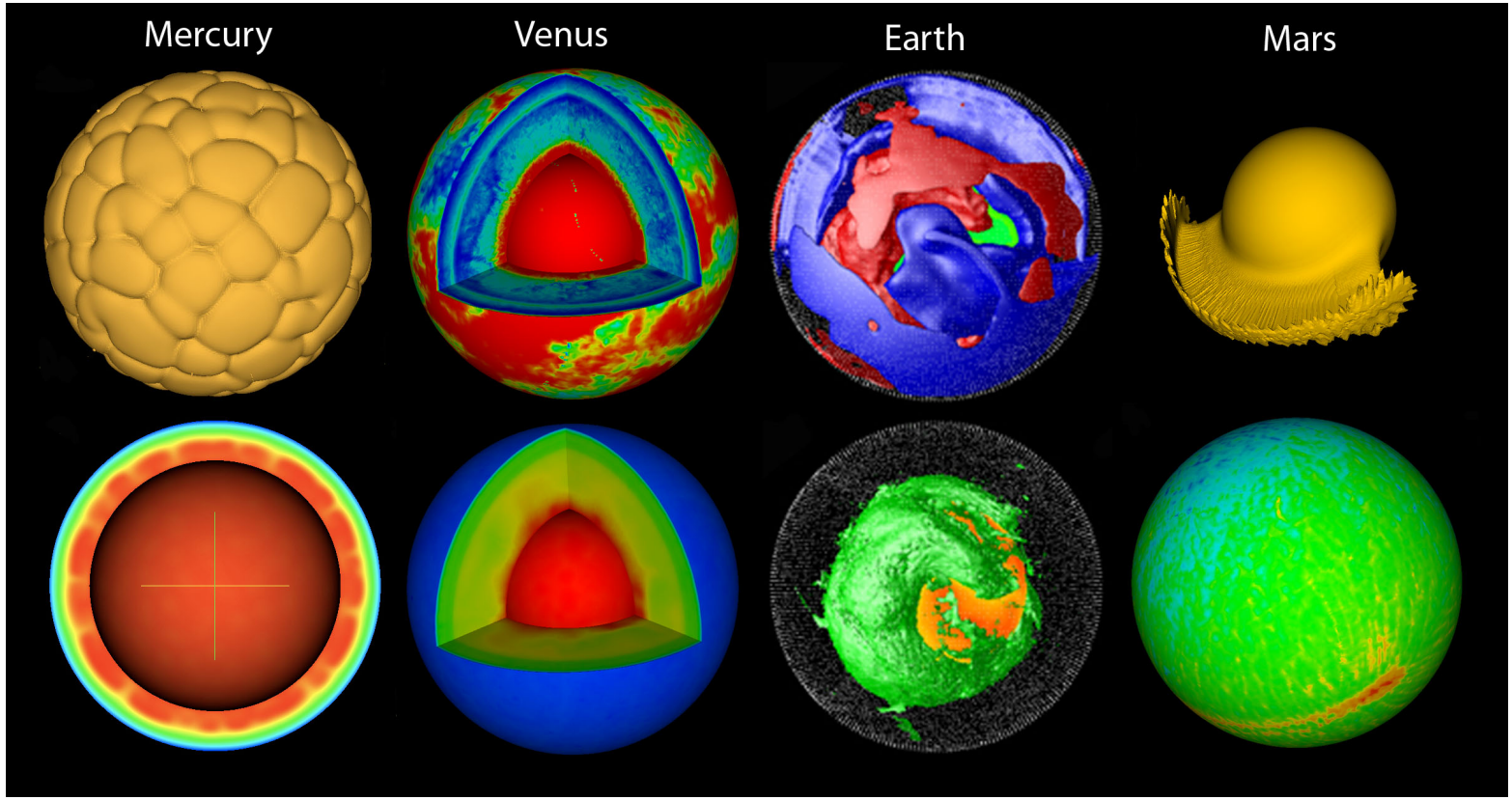
Smooth 16x16x16

Smooth 8x8x8

Exact solution 4x4x4



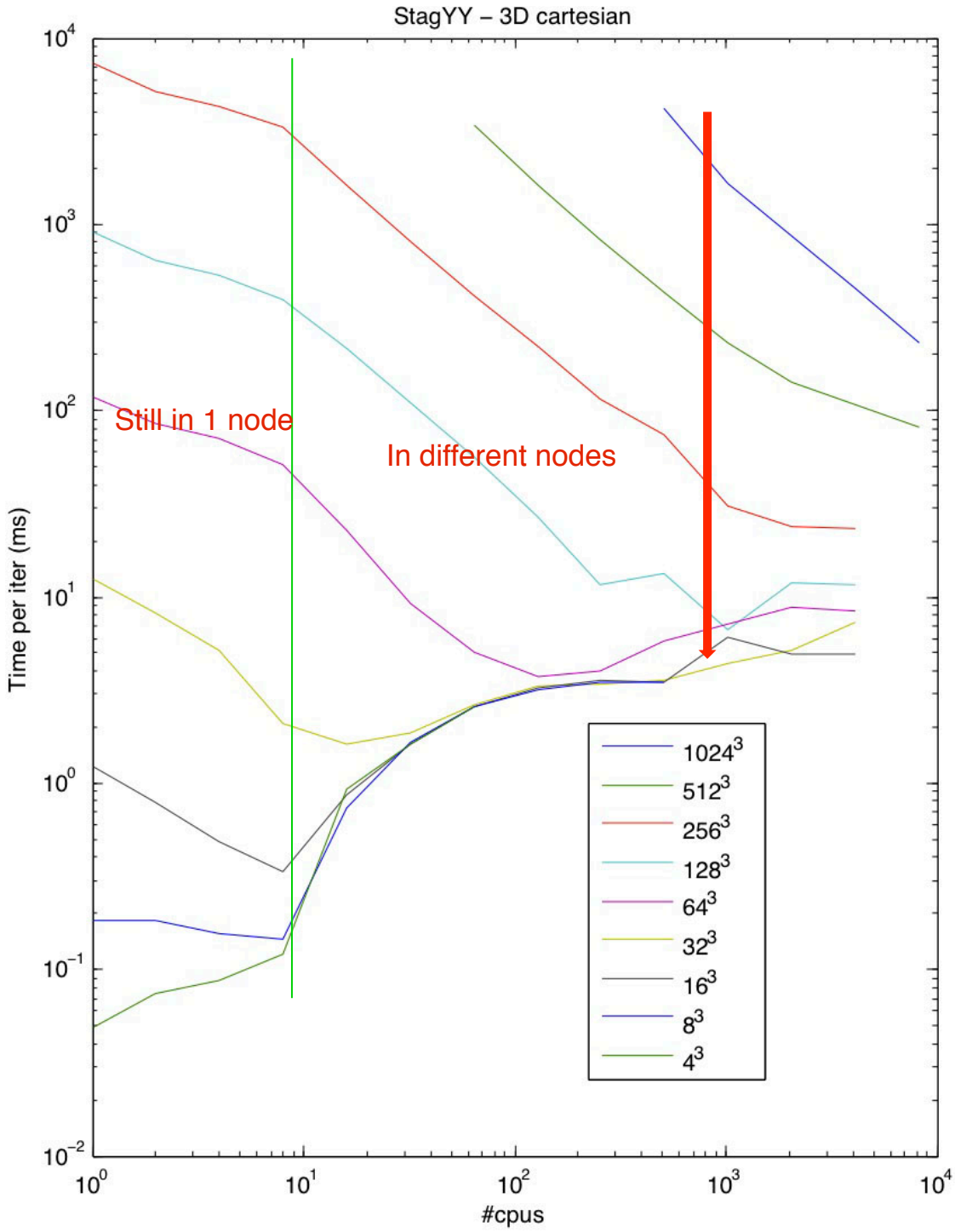
Application to StagYY
Cartesian or spherical



StagYY iterations: 3D Cartesian

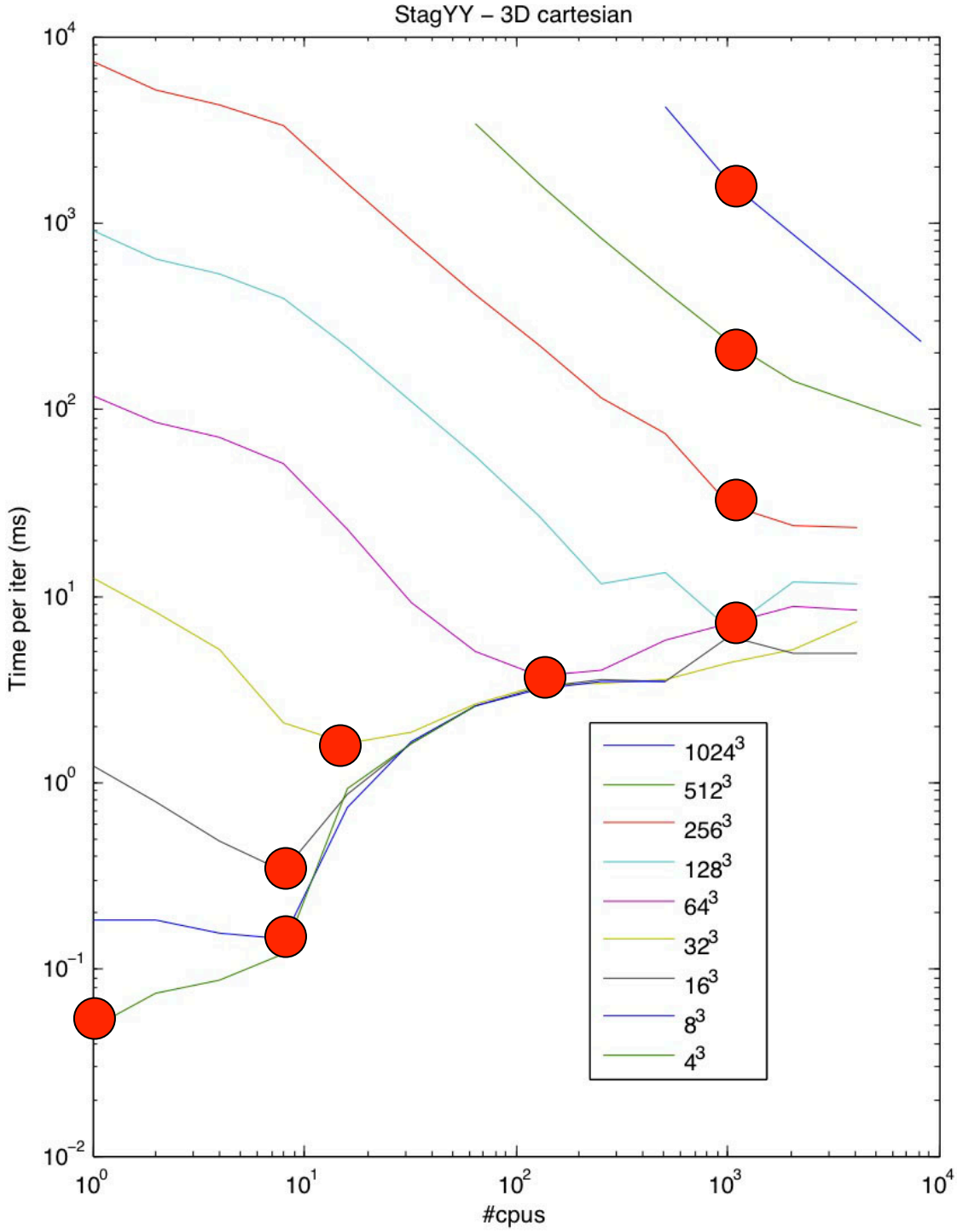
Change in scaling from same-node to cross-node communication

Simple-minded multigrid:
Very inefficient coarse levels!
Exact coarse solution can take long time!

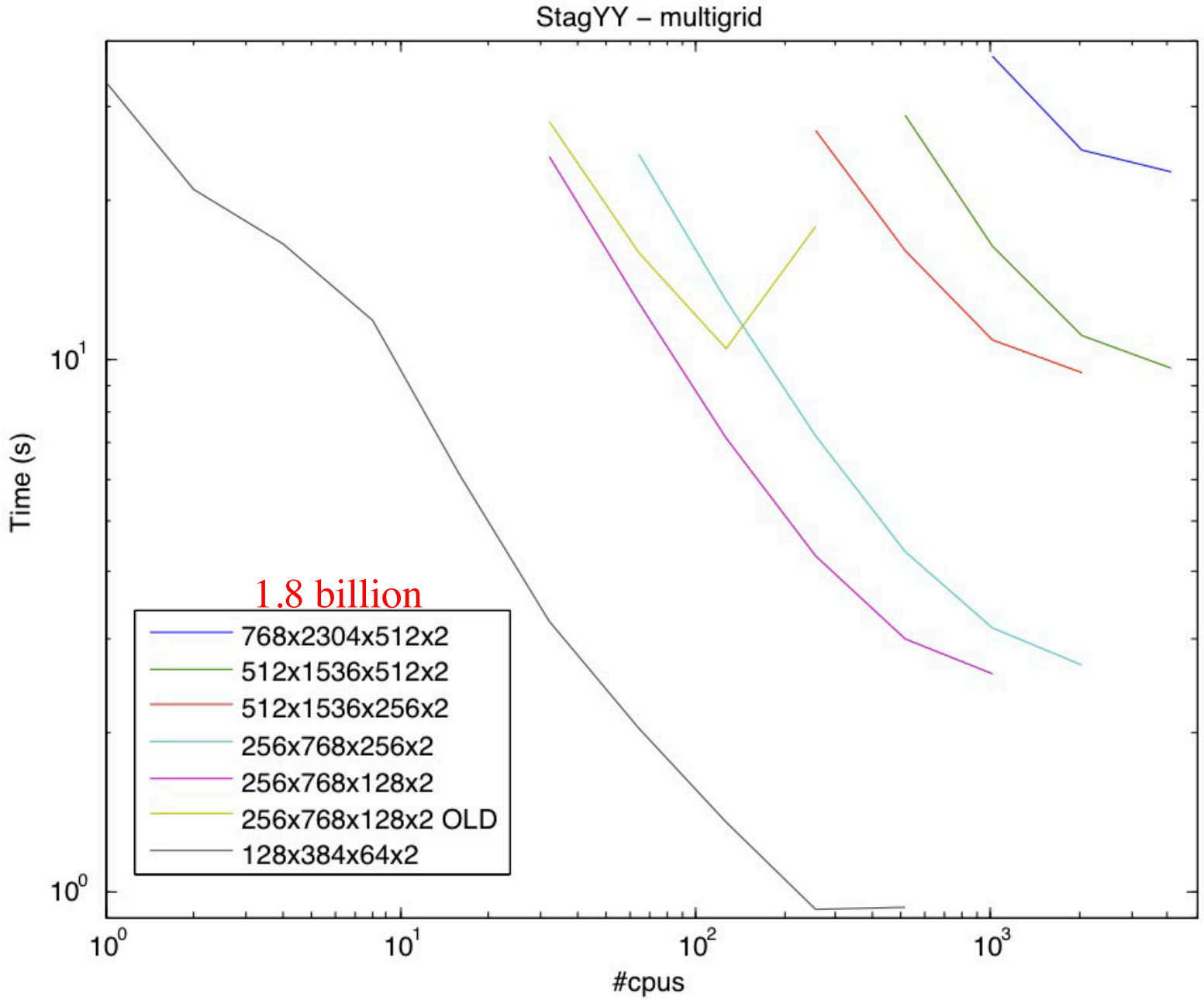


New treatment: follow minima

- Keep #points/core > minimum (tuned for system)
- Different for on-node and cross-node communication



Multigrid – now (& before): yin-yang



Summary

- For very large-scale problems, need to parallelise code using MPI
- For finite-difference codes, the best method is to assign different parts of the domain to different CPUs ("domain decomposition")
- The code looks similar to before, but with some added routines to take care of communication
- Multigrid scales fine on 1000s CPUs if:
  - Treat coarse grids on subsets of CPUs
  - Large enough total problem size

For more information

- https://computing.llnl.gov/tutorials/parallel_comp/
- http://en.wikipedia.org/wiki/Parallel_computing
- http://www.mcs.anl.gov/~itf/dbpp/
- http://en.wikipedia.org/wiki/Message_Passing_Interface