Numerical Modelling in FORTRAN day 10

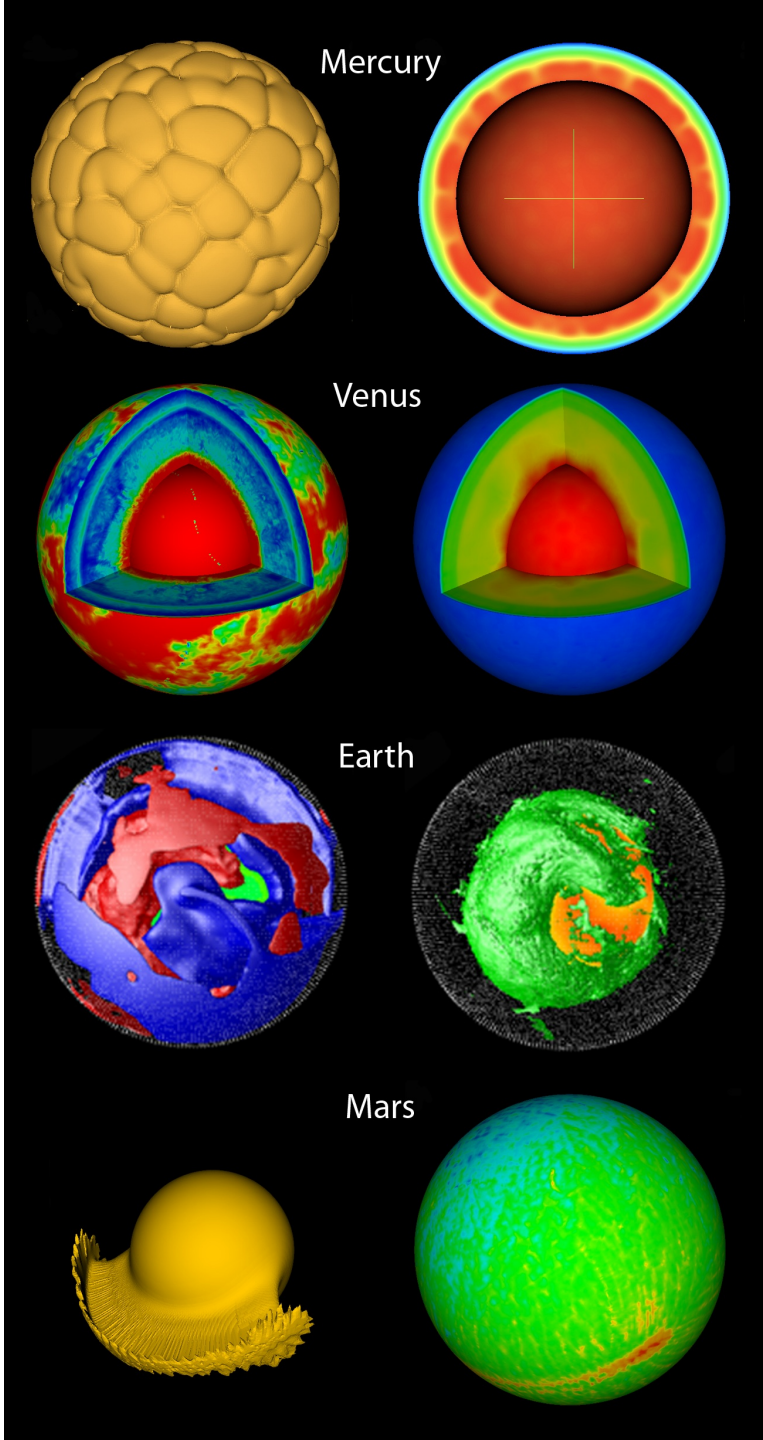
Paul Tackley, 2020

Today‘s Goals

1. Learn about parallel computing and how Fortran programs can be parallelized.
2. Implicit time stepping for diffusion.

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

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



Huge problems
=> huge computer

www.top500.org

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	**JUWELS Booster Module** - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
9	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
10	**Dammam-7** - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 SXM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6	

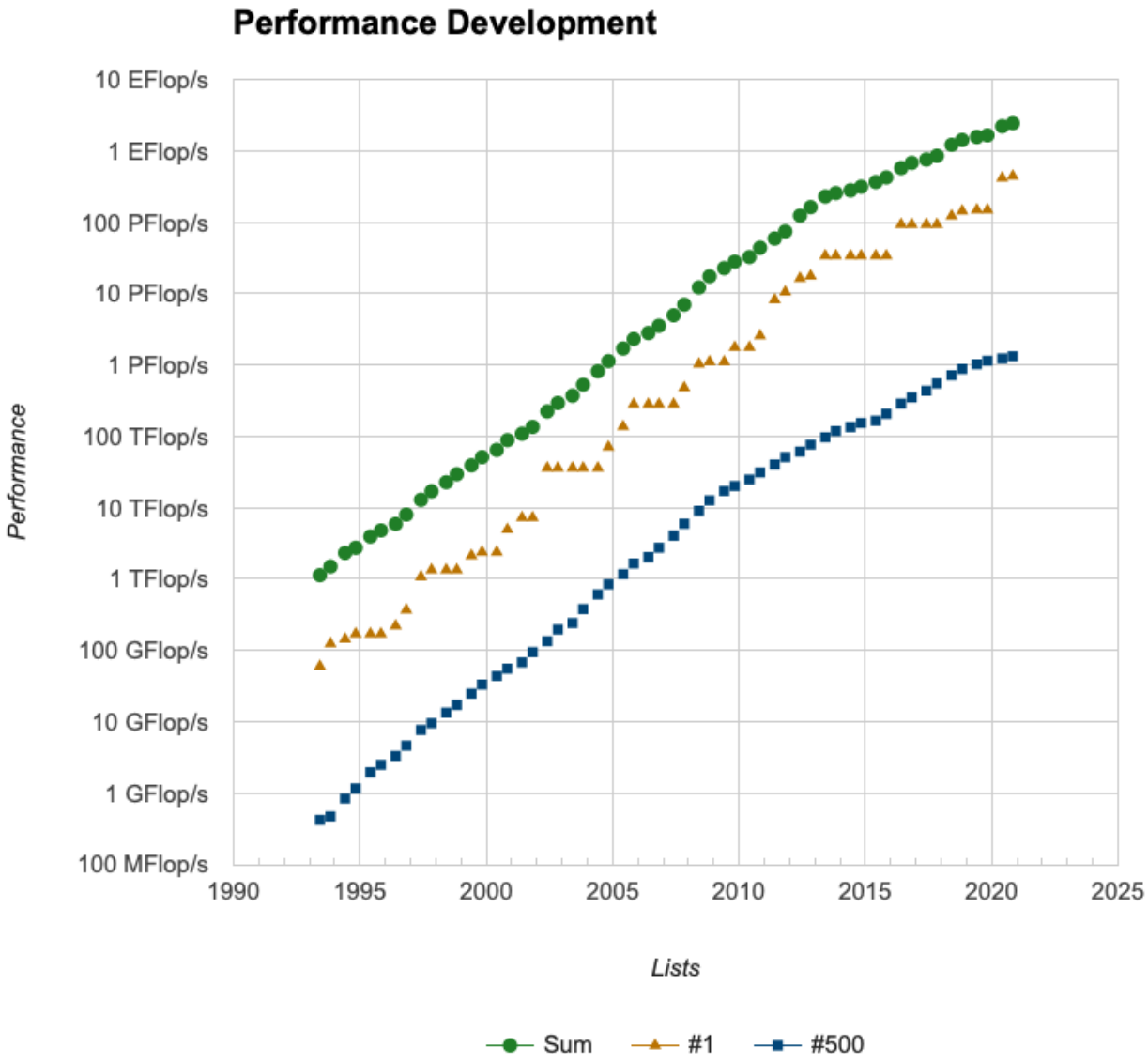
Huge problems
=> huge computer

www.top500.org

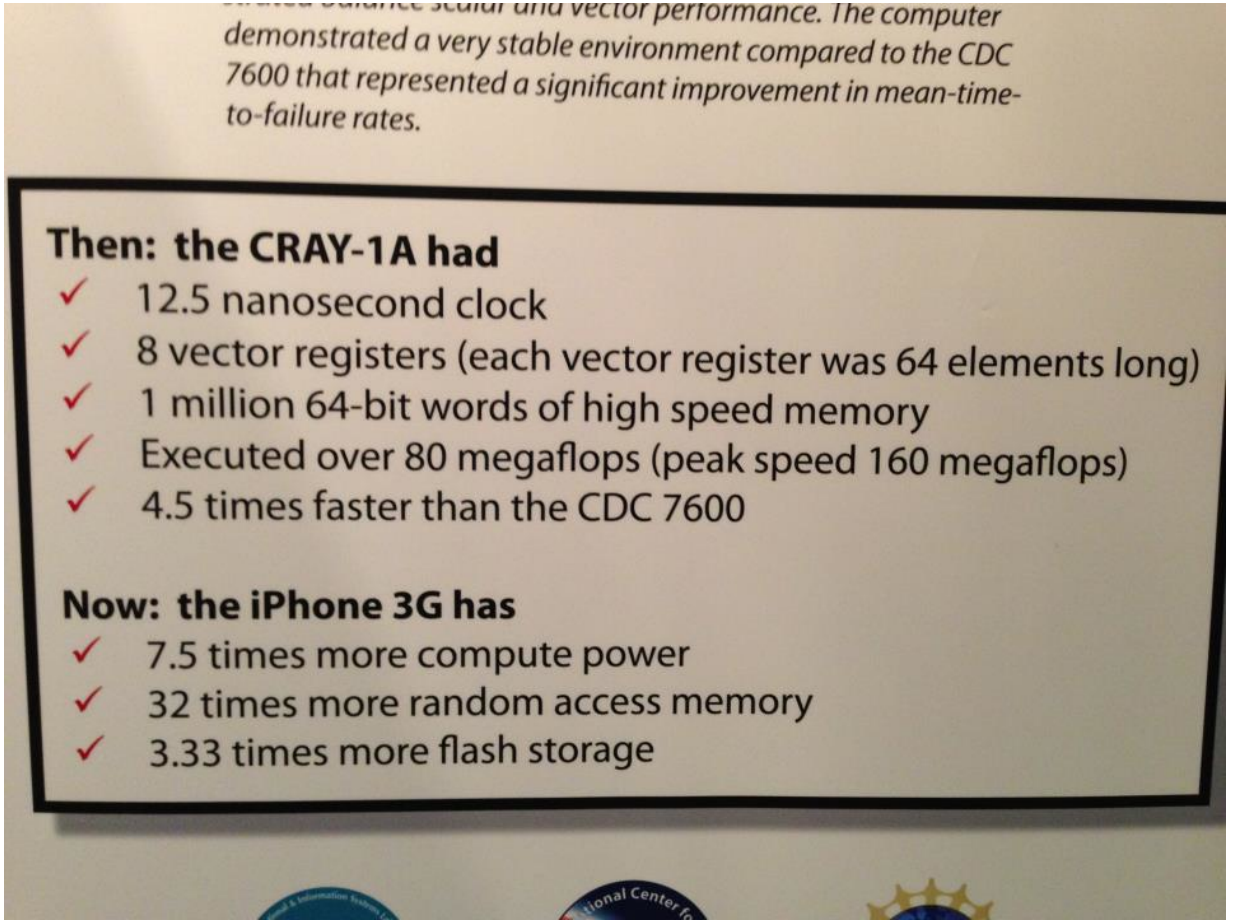
Trends

- Up to millions of cores
- Many use GPUs

11	**Marconi-100** - IBM Power System AC922, IBM POWER9 16C 3GHz, Nvidia Volta V100, Dual-rail Mellanox EDR Infiniband, IBM CINECA Italy	347,776	21,640.0	29,354.0	1,476
12	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100, Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
13	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect, Cray/HPE DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578
14	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR, Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
15	**SuperMUC-NG** - ThinkSystem SD650, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path, Lenovo Leibniz Rechenzentrum Germany	305,856	19,476.6	26,873.9	
16	**Hawk** - Apollo 9000, AMD EPYC 7742 64C 2.25GHz, Mellanox HDR Infiniband, HPE HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	698,880	19,334.0	25,159.7	3,906
17	**Lassen** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Tesla V100, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	288,288	18,200.0	23,047.2	
18	**PANGEA III** - IBM Power System AC922, IBM POWER9 18C 3.45GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100, IBM Total Exploration Production France	291,024	17,860.0	25,025.8	1,367
19	**TOKI-SORA** - PRIMEHPC FX1000, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu Japan Aerospace eXploration Agency Japan	276,480	16,592.0	19,464.2	
20	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray/HPE DOE/SC/LBNL/NERSC United States	622,336	14,014.7	27,880.7	3,939

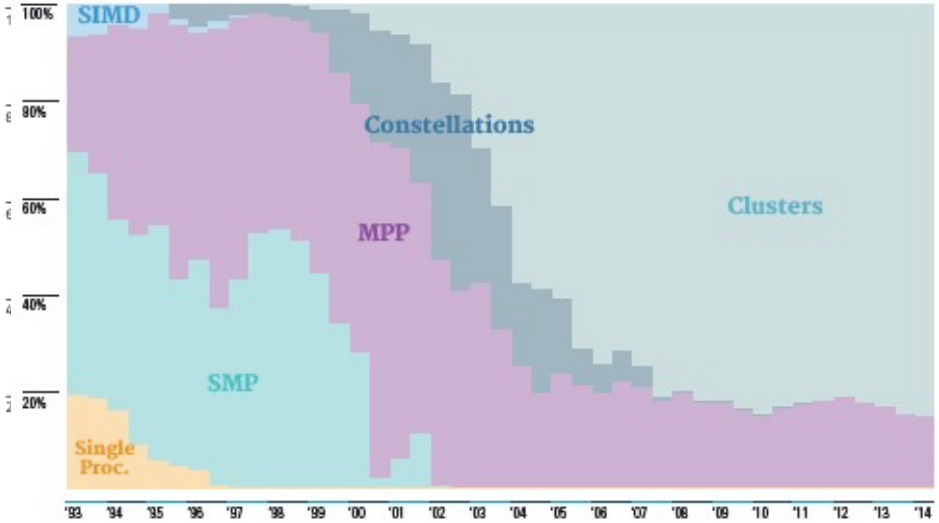

Performance Development
10 EFlop/s
1 EFlop/s
100 PFlop/s
10 PFlop/s
1 PFlop/s
100 TFlop/s
10 TFlop/s
1 TFlop/s
100 GFlop/s
10 GFlop/s
1 GFlop/s
100 MFlop/s
Performance
1990
1995
2000
2005
2010
2015
2020
2025
Lists
Sum
#1
#500

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

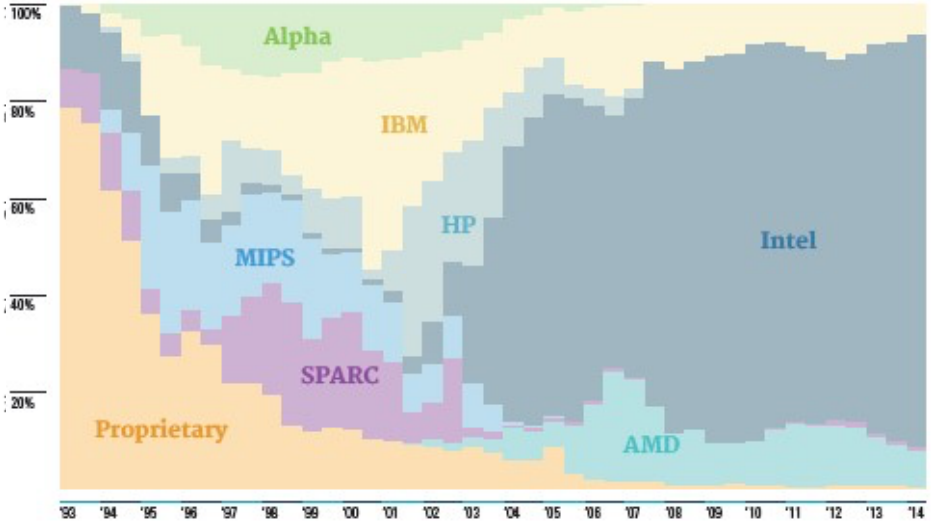


(photo taken at NCAR museum)

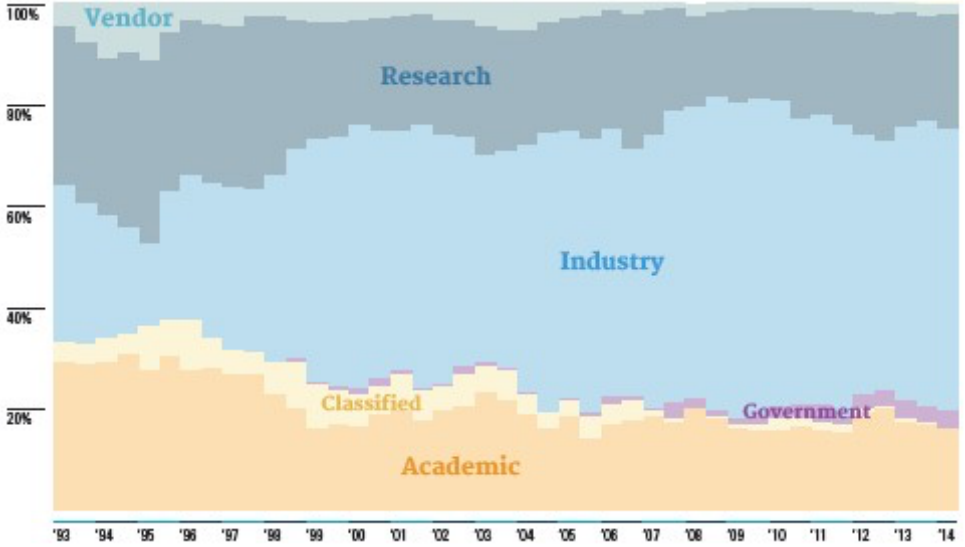
ARCHITECTURES



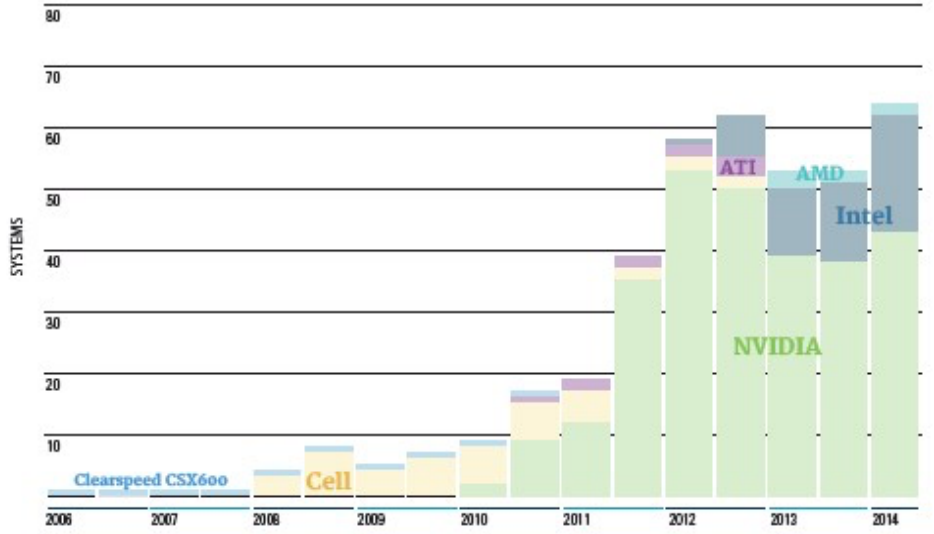
CHIP TECHNOLOGY



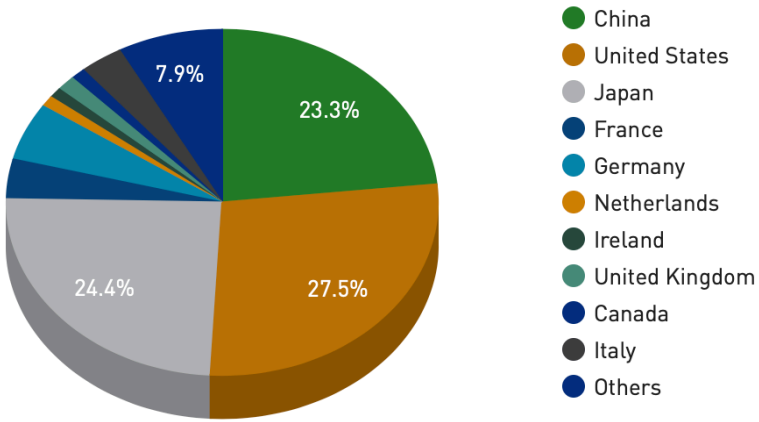
INSTALLATION TYPE



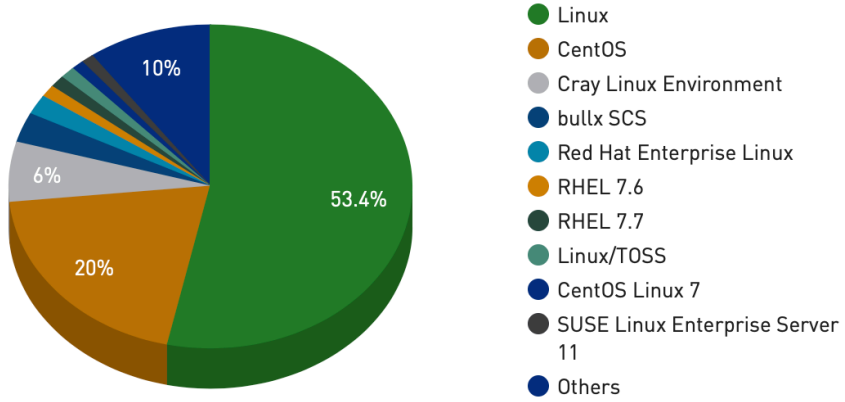
ACCELERATORS / CO-PROCESSORS



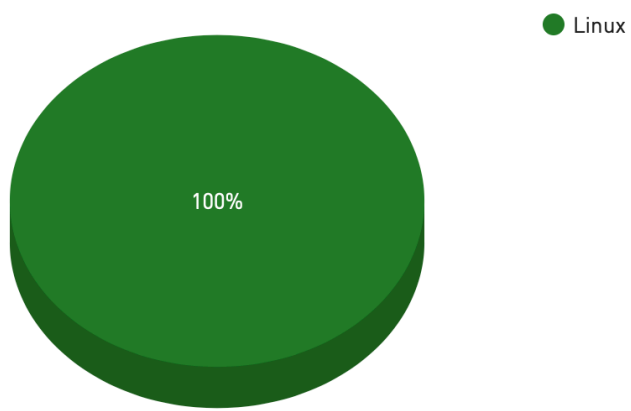
Countries Performance Share



Operating System System Share



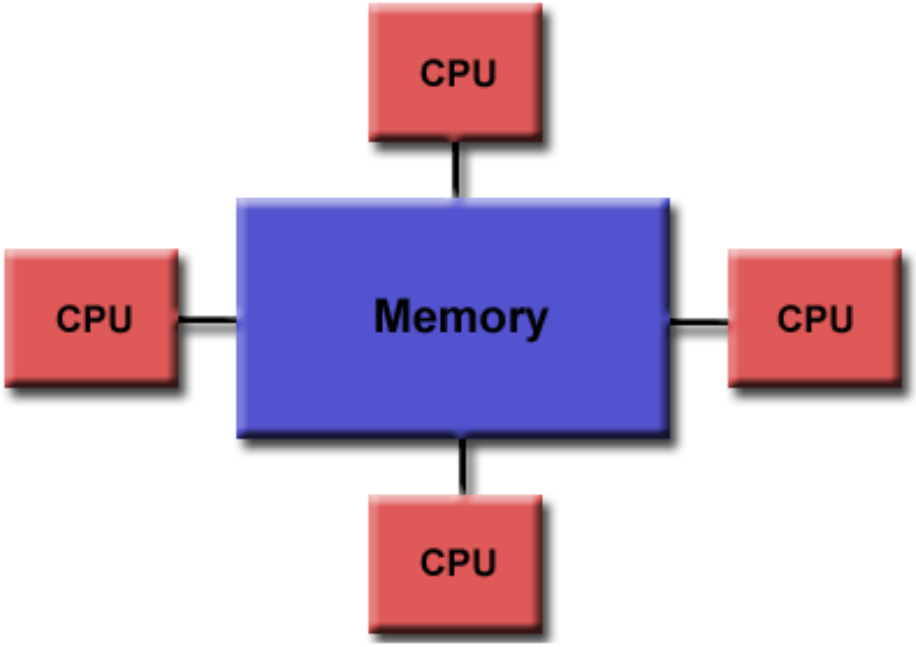
Operating system Family System Share



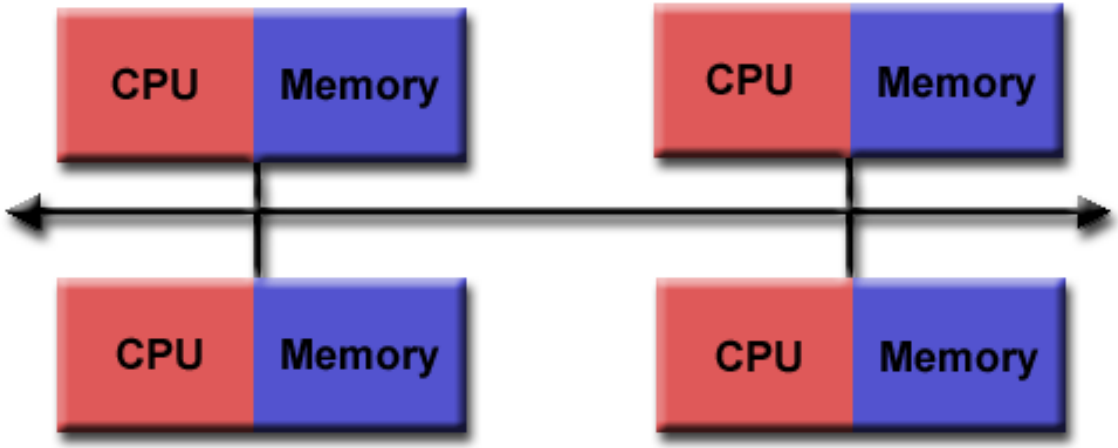
In Switzerland

12	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100, Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384

Each node: 12-core Intel CPU + GPU



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)

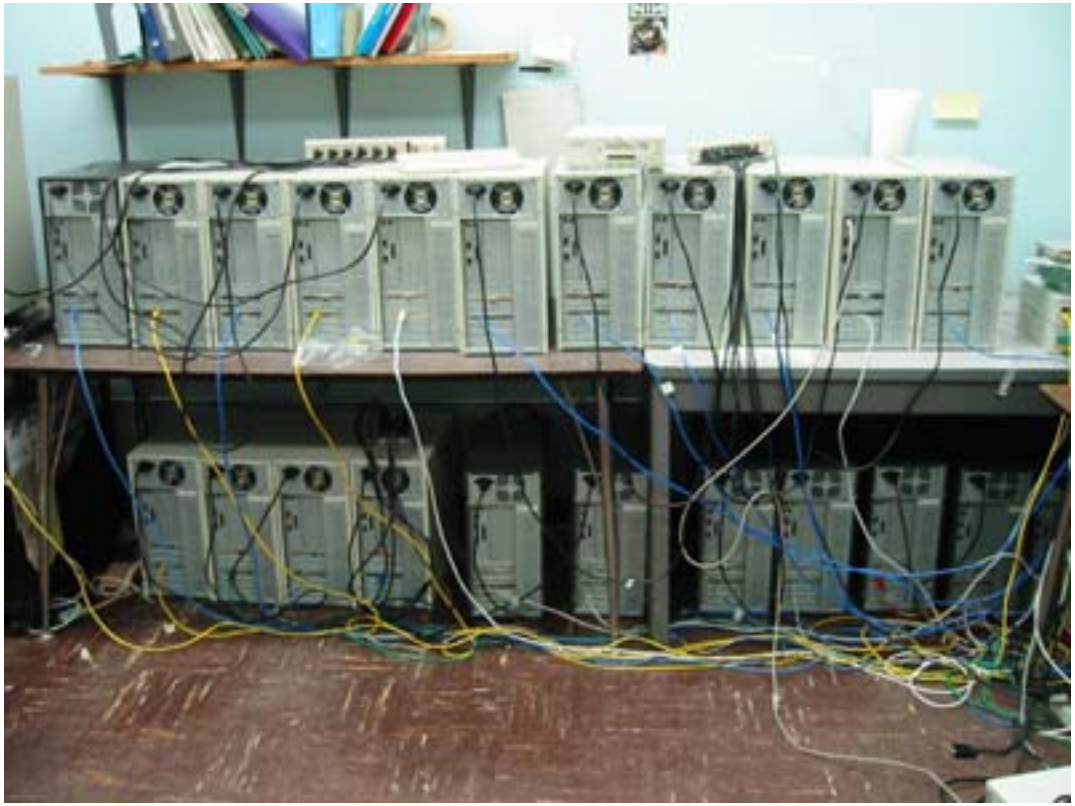
The current #1: Fugaku

Fugaku *(Wikipedia)*

Active	From 2021
Sponsors	MEXT
Operators	RIKEN
Location	RIKEN Center for Computational Science (R-CCS)
Architecture	158,976 nodes Fujitsu A64FX CPU (48+4 core) per node Tofu interconnect D
Operating system	Custom Linux-based kernel
Memory	HBM2 32 GiB/node
Storage	1.6 TB NVMe SSD/16 nodes (L1) 150 PB shared Lustre FS (L2)[1] Cloud storage services (L3)
Speed	442 PFLOPS (per TOP500 Rmax), after upgrade; higher 2.0 EFLOPS on a different mixed-precision benchmark
Cost	US$1 billion (total programme cost)[2][3]
Ranking	TOP500: 1, June 2020
Web site	www.r-ccs.riken.jp/en/fugaku
Sources	Fugaku System Configuration

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

Using standard PC cases:



or using rack-mounted cases

Programming approaches

- OpenMP: Shared memory only (same node)
- MPI (Message-Passing-Interface): Any number of cores, distributed memory
- Coarray Fortran. Since Fortran 2008, with 2018 extensions.
- CUDA Fortran. For GPUs.

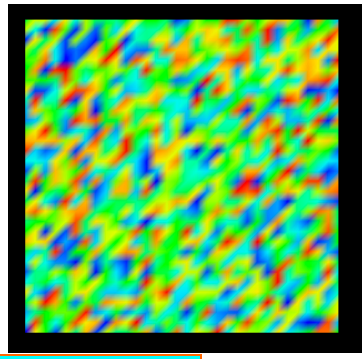
MPI: Message-Passing Interface

- 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:
  - https://www.mpich.org
  - http://www.open-mpi.org/

How to parallelise a code: worked example
Use MPI first, then Fortran coarrays

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:

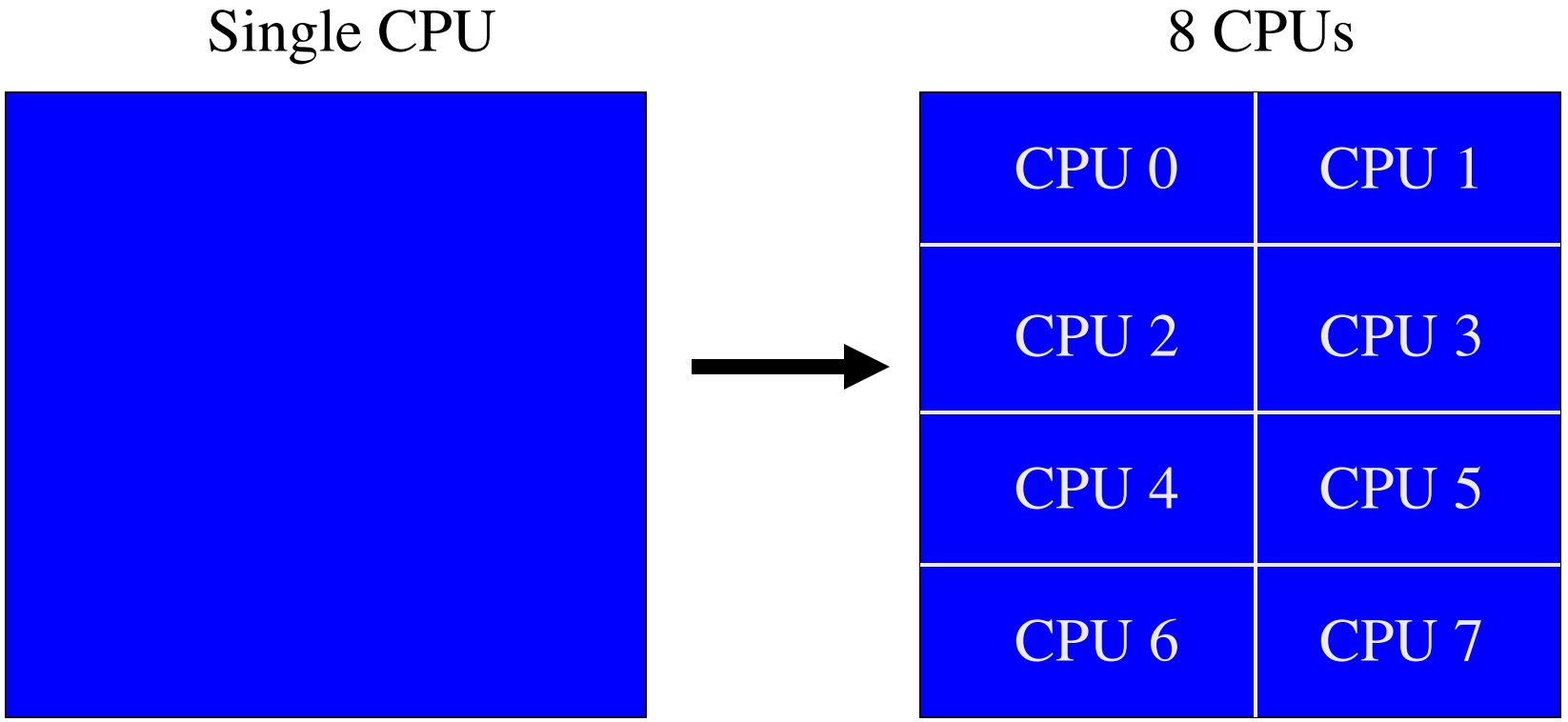
$$\tilde{u}_{ij}^{n+1} = \tilde{u}_{ij}^{n} + \alpha R_{ij}\frac{h^2}{6}$$

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



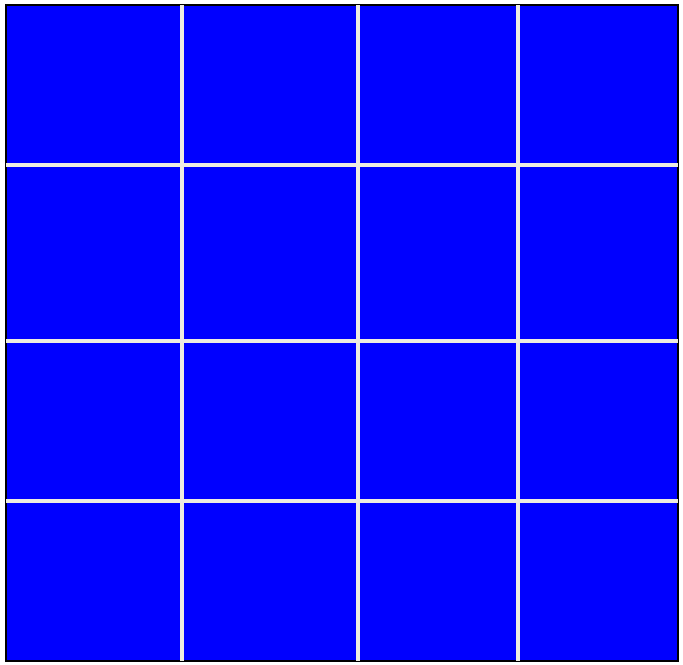
Each CPU will do the same operations but on different parts of the domain

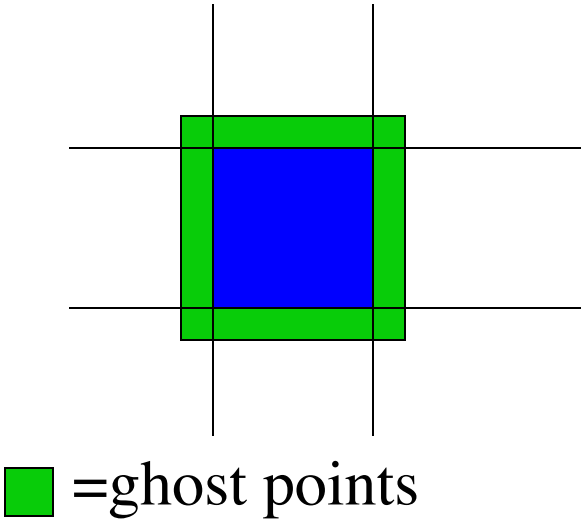
You need to build parallelization into the code using MPI

- 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)
  - Global combinations (MPI_allreduce)
  - CPU-CPU communication (MPI_send,MPI_recv…)

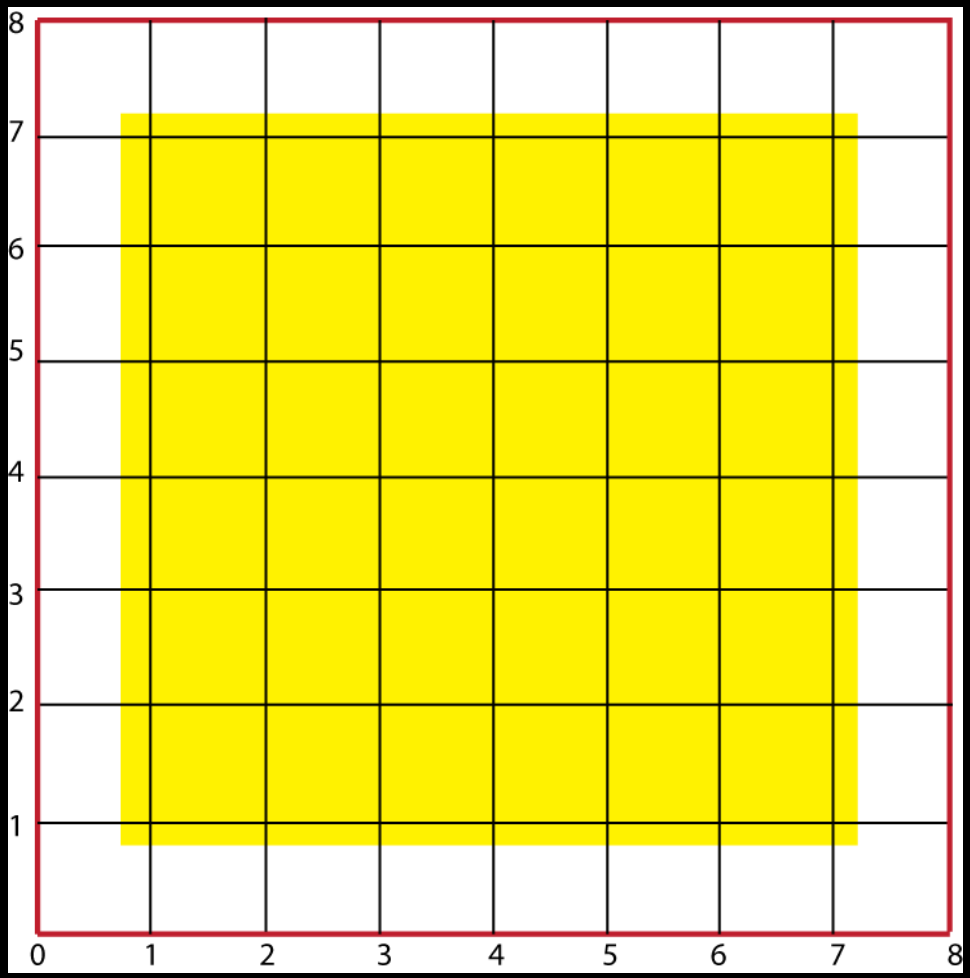
Boundaries

- 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





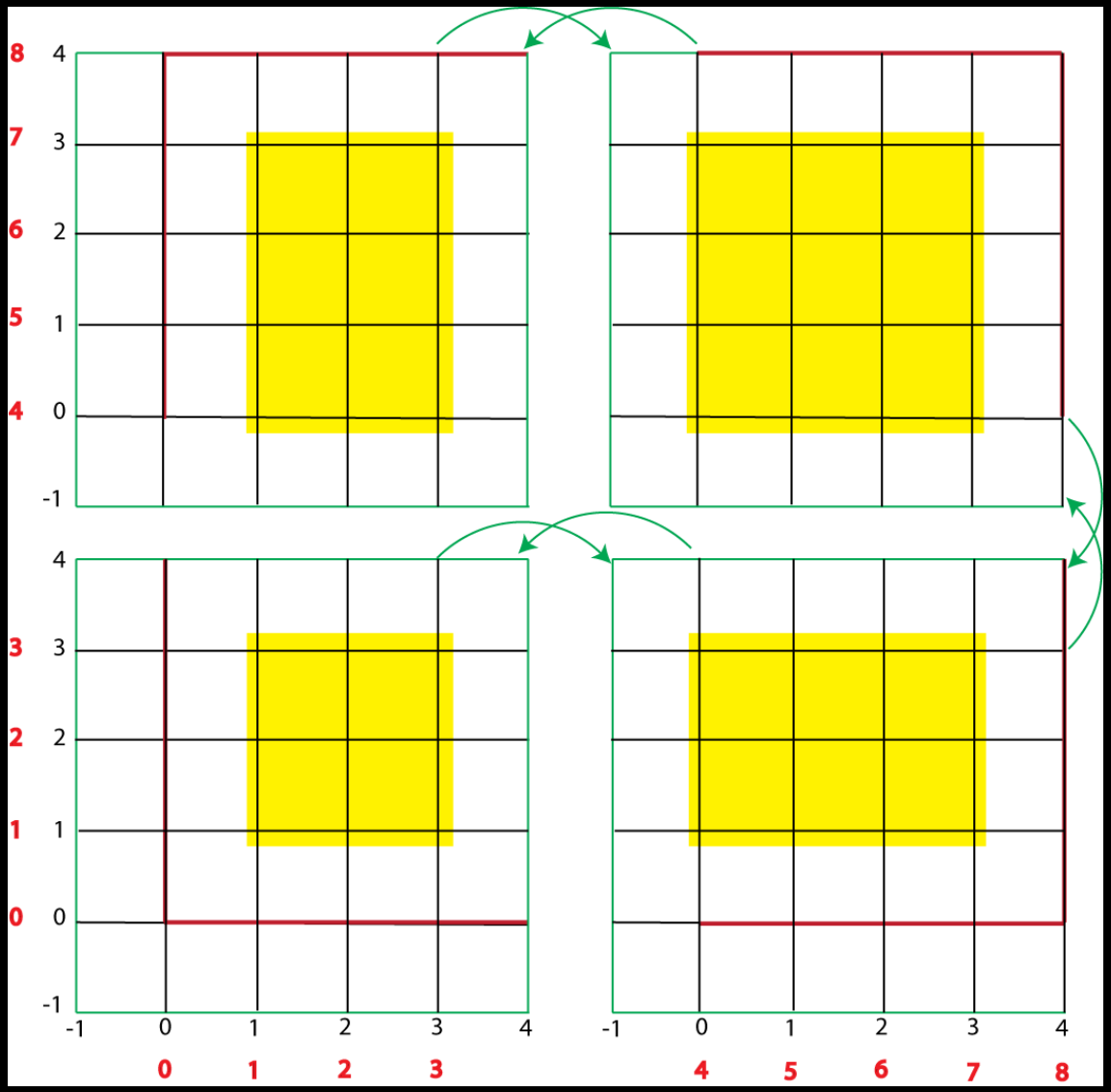
Scalar Grid



Red=boundary points (=0)

Yellow=iterated/solved (1…n-1)

Parallel grids



Red=ext. boundaries

Green=int. boundaries

Yellow=iterated/solved

First things the code has to do:

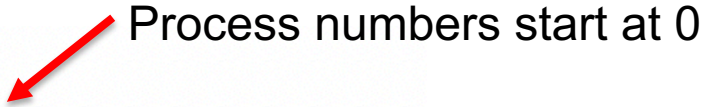
- Call MPI_init(ierr)
- Find #CPUs using MPI_com_size
- Find which CPU it is, using MPI_com_rank (returns a number from 0…#CPUs-1)
- Calculate which part of the global grid it is dealing with, and which other CPUs are handling neighboring subdomains.

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
```

Example: “Hello world” program

```
[£ mpif90 HelloMPI.f90
[£ mpirun -n 8 a.out
 Hello from cpu number :           0  of           8
 Hello from cpu number :           1  of           8
 Hello from cpu number :           2  of           8
 Hello from cpu number :           3  of           8
 Hello from cpu number :           5  of           8
 Hello from cpu number :           6  of           8
 Hello from cpu number :           4  of           8
 Hello from cpu number :           7  of           8
```

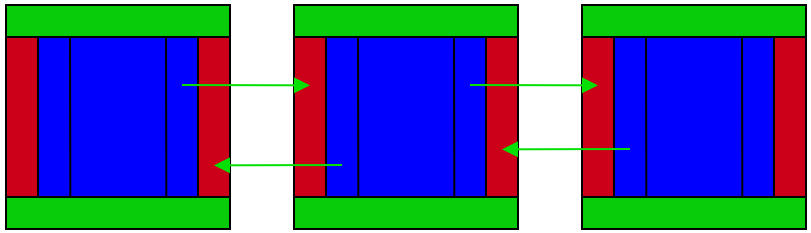


Moving forward

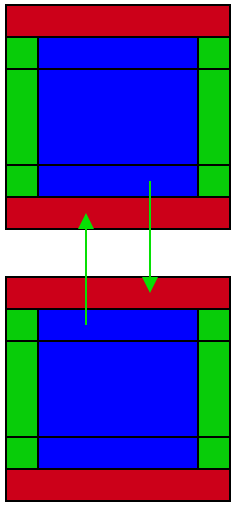
- Update values in subdomain using ‘ghost points’ as boundary condition, i.e.,
  - Timestep (explicit), or
  - Iteration (implicit)
- Update ghost points by communicating with other CPUs
- Works well for explicit or iterative approaches

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=64,ny=64,nz=64                  ! #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=64,ny=64,nz=64                    ! #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
```

```
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,status(MPI_status_size)
  call MPI_sendrecv (sendbuf,length,MPI_real,othercpu,mycpu, &
       recvbuf,length,MPI_real,othercpu,othercpu,MPI_comm_world,status,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
```

Not optimal – uses blocking send/receive

Better: using non-blocking (isend/irecv)

```
subroutine update_sides()
  integer m(2),ierr
  call sides1d(npx,  &                  ! x direction
       myx>0    ,u(0    ,:,:),u(-1 ,:,:),cpu_xm, &
       myx<ncx-1,u(nxl-1,:,:),u(nxl,:,:),cpu_xp )
  call sides1d(npy,  &                  ! y direction
       myy>0    ,u(:,    0,:),u(:, -1,:),cpu_ym,  &
       myy<ncy-1,u(:,nyl-1,:),u(:,nyl,:),cpu_yp )
  call sides1d(npz,  &                  ! z direction
       myz>0    ,u(:,:,    0),u(:,:, -1),cpu_zm,  &
       myz<ncz-1,u(:,:,nzl-1),u(:,:,nzl),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
```

Fortran Coarrays

- Fortran 2008; extended in Fortran 2018
- Each process (“image”) has its own version of the coarray
- Each image can access the versions on other images using []

Fortran Coarrays

Always *. Actual limit is set at run time

```
real, dimension(10), codimension[*] :: x

real :: y(10)[*]

real :: z(10)[0:3,*]

x(:) = y(:)[ni]
```

Alternative syntax

Multidimensional codimensions

Sets local x(:) equal to y(:) on image ni.

Example: “Hello world” program

Intrinsic functions

```
program HelloCoarray

  print*,'Hello from cpu number :',this_image(),' of ',num_images()

end program HelloCoarray
```

Image numbers start at 1

```
[£ caf HelloCoarray.f90
[£ cafrun -np 4 a.out
  Hello from cpu number :           1  of           4
  Hello from cpu number :           2  of           4
  Hello from cpu number :           3  of           4
  Hello from cpu number :           4  of           4
```

```
program SimplePoissonCoarray

  implicit none

  integer:: nx=64,ny=64,nz=64                    ! #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)

  ! 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)[*])
  sync all
  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 co_max(resmax)
     if (mycpu==1) print*,iter,resmax

  end do

contains
```

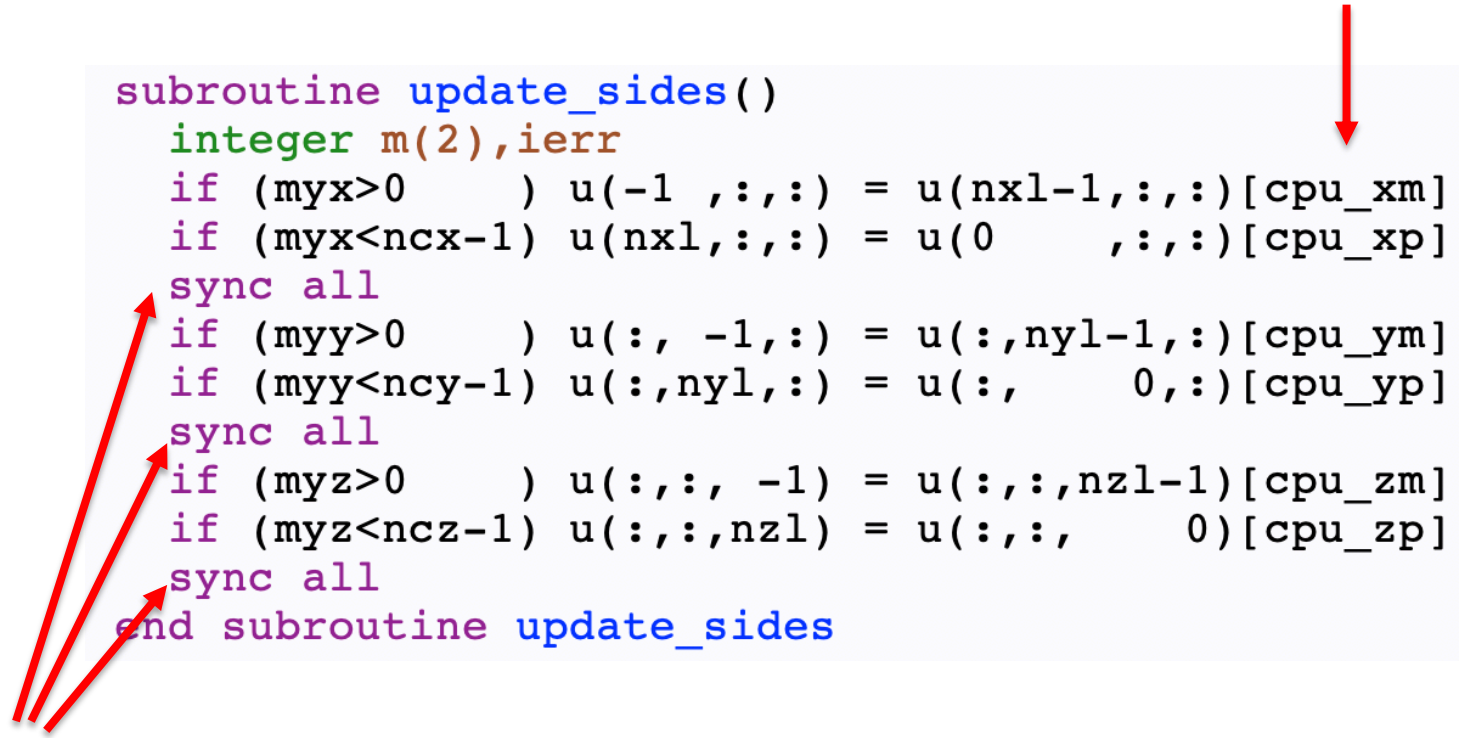
Use [:] when declaring

Use [*] when allocating

No [] means local version

Intrinsic global maximum

Coarrays greatly simplify the communication!



```
subroutine update_sides()
  integer m(2),ierr
  if (myx>0    ) u(-1 ,:,:) = u(nxl-1,:,:)[cpu_xm]
  if (myx<ncx-1) u(nxl,:,:) = u(0    ,:,:)[cpu_xp]
  sync all
  if (myy>0    ) u(:, -1,:) = u(:,nyl-1,:)[cpu_ym]
  if (myy<ncy-1) u(:,nyl,:) = u(:,    0,:)[cpu_yp]
  sync all
  if (myz>0    ) u(:,:, -1) = u(:,:,nzl-1)[cpu_zm]
  if (myz<ncz-1) u(:,:,nzl) = u(:,:,    0)[cpu_zp]
  sync all
end subroutine update_sides
```

sync all waits for communication to finish before continuing.
Important otherwise the calculation might be using values that have not yet arrived

Coarray commands & functions (f2008)

- this_image(), num_images()
- sync all, sync images(list), sync memory
- stop all
- co_sum(), co_broadcast(), co_min(), co_max(), co_reduce() (f2018)
- lcobound(), ucobound()
- lock & unlock: lock variable by an image
- Critical & end critical: block of code executed by one image at a time

Coarrays: compiler status

- ifort: built in, except on MacOS
- gfortran: single image compilation is built in (gfortran –fcoarray=single); for parallel execution additional package(s) must be installed
  - **OpenCoarrays** provides a convenient wrapper around gfortran & MPI (see *Hello World* example)

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

$$\nabla^2 u = f$$

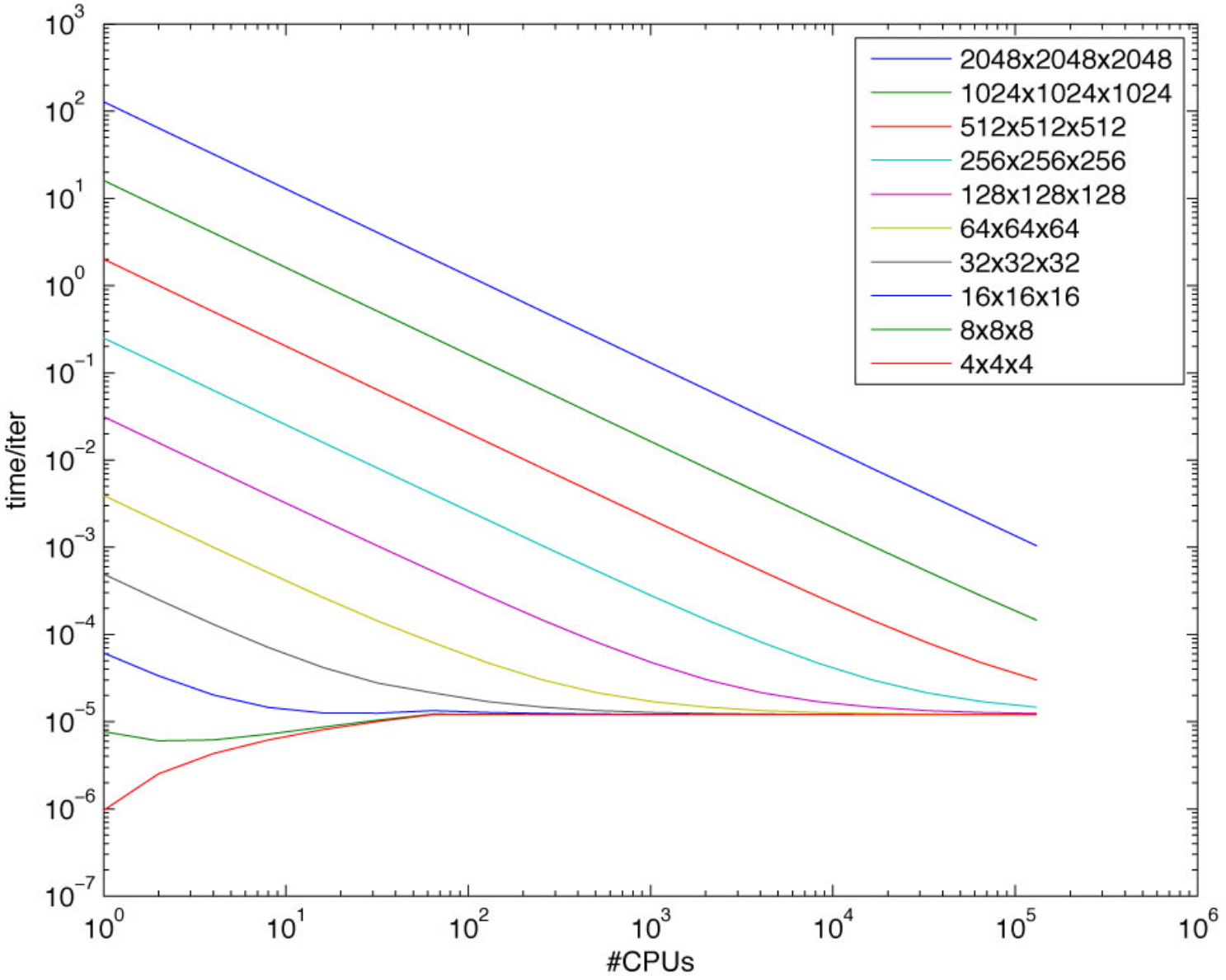
$$t = aN^3 + nL + bN^2/B$$

Assume 15 operations/point/iteration & 1 Gflop performance
$\Rightarrow$a=15/1e9=1.5e-8

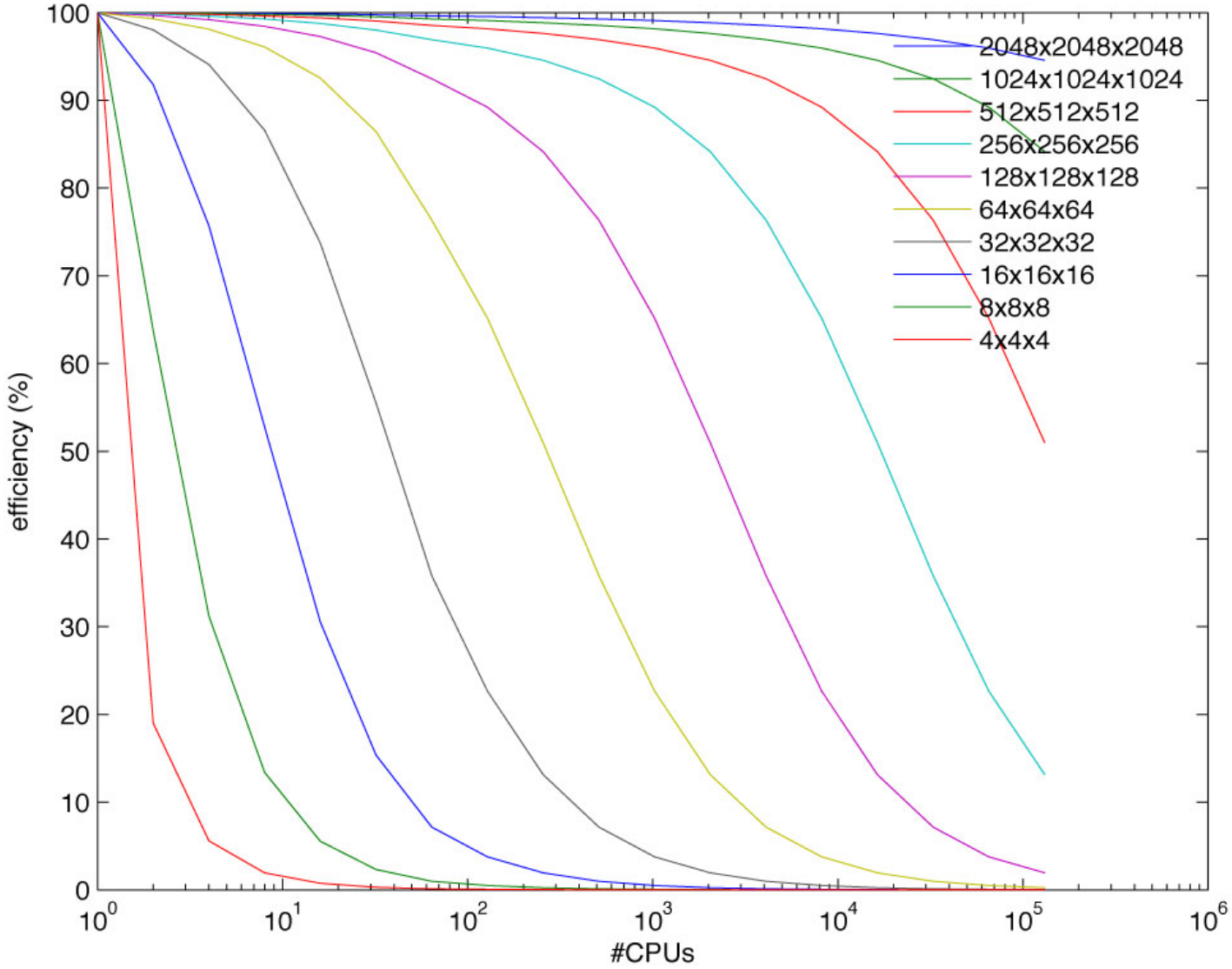
If 3D decomposition, n=6, b=6*4 (single precision)

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

Up to 2e5 CPUs (Quadrics communication)



Efficiency


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

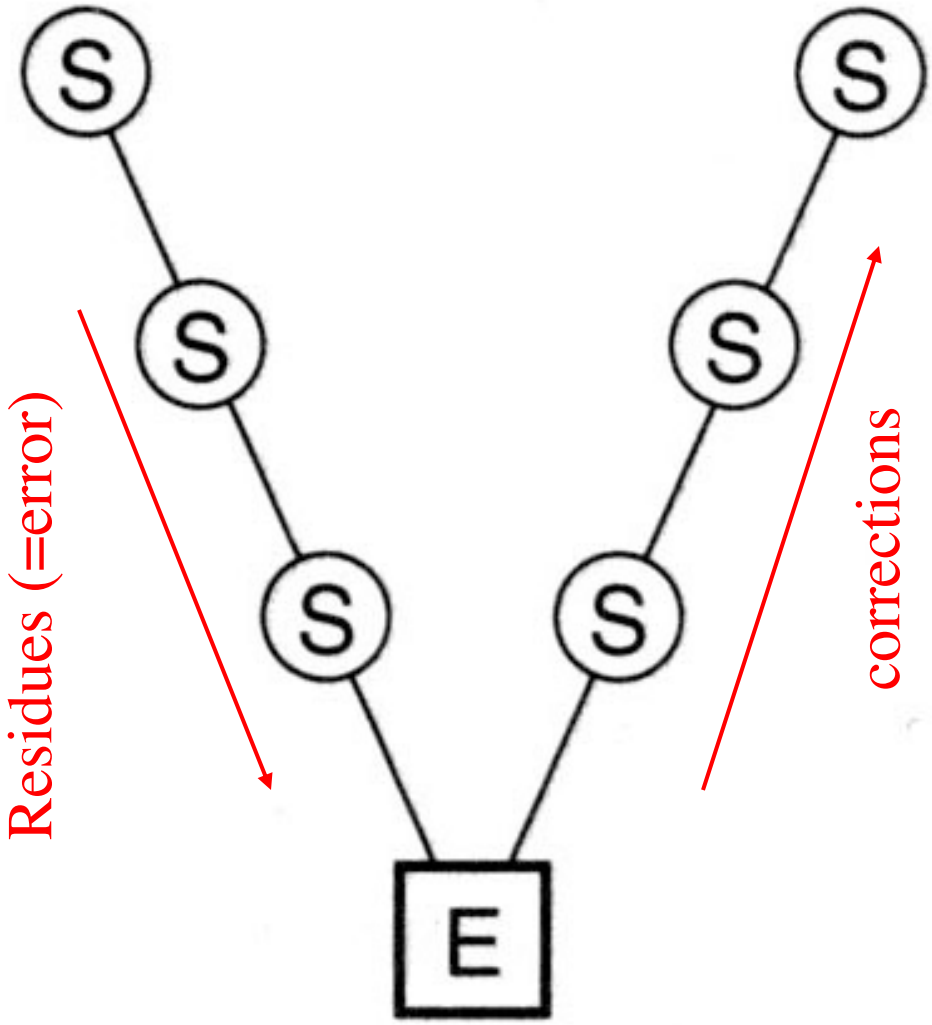
Now multigrid V cycles

Smooth 32x32x32

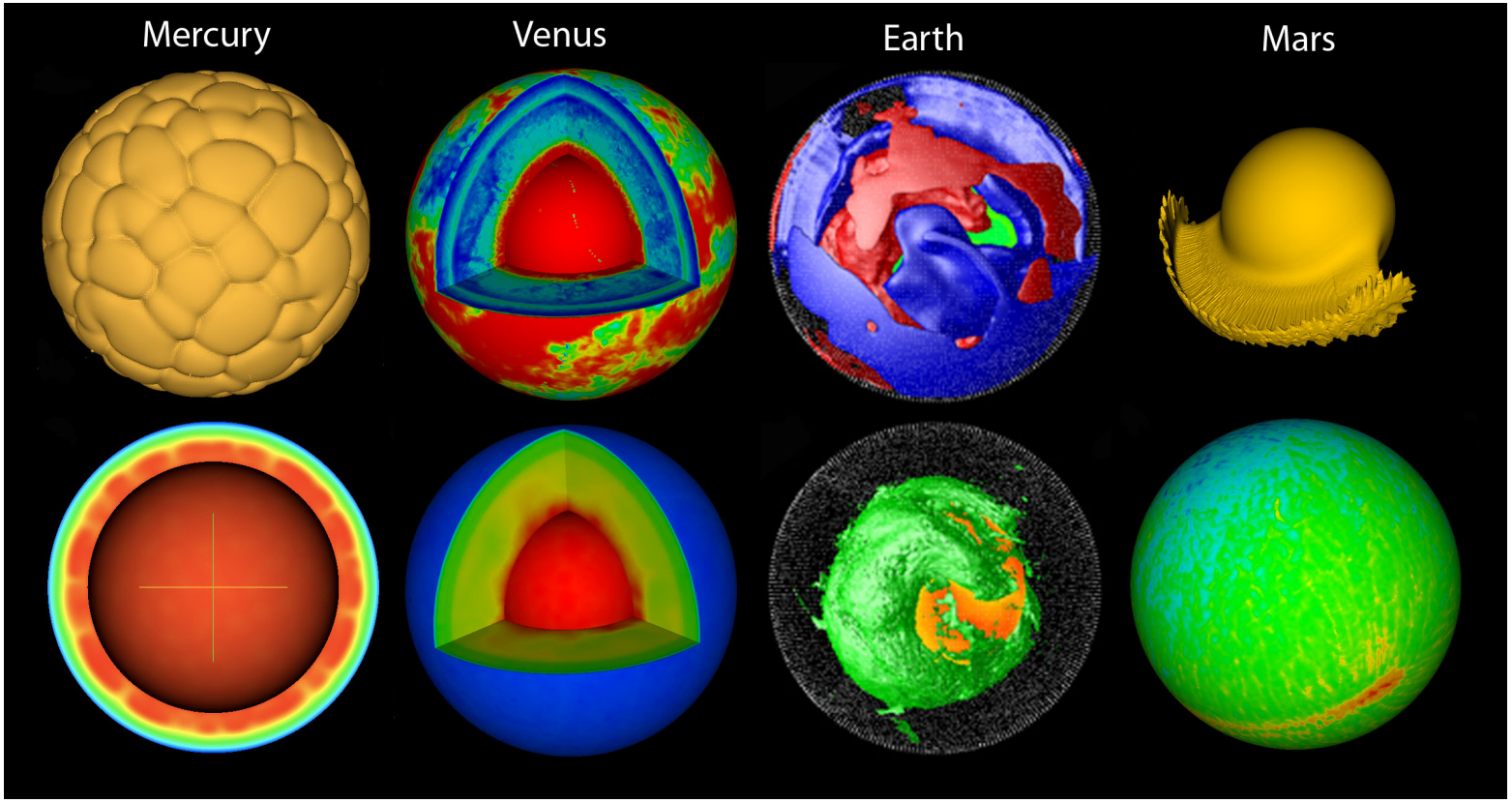
Smooth 16x16x16

Smooth 8x8x8

Exact solution 4x4x4



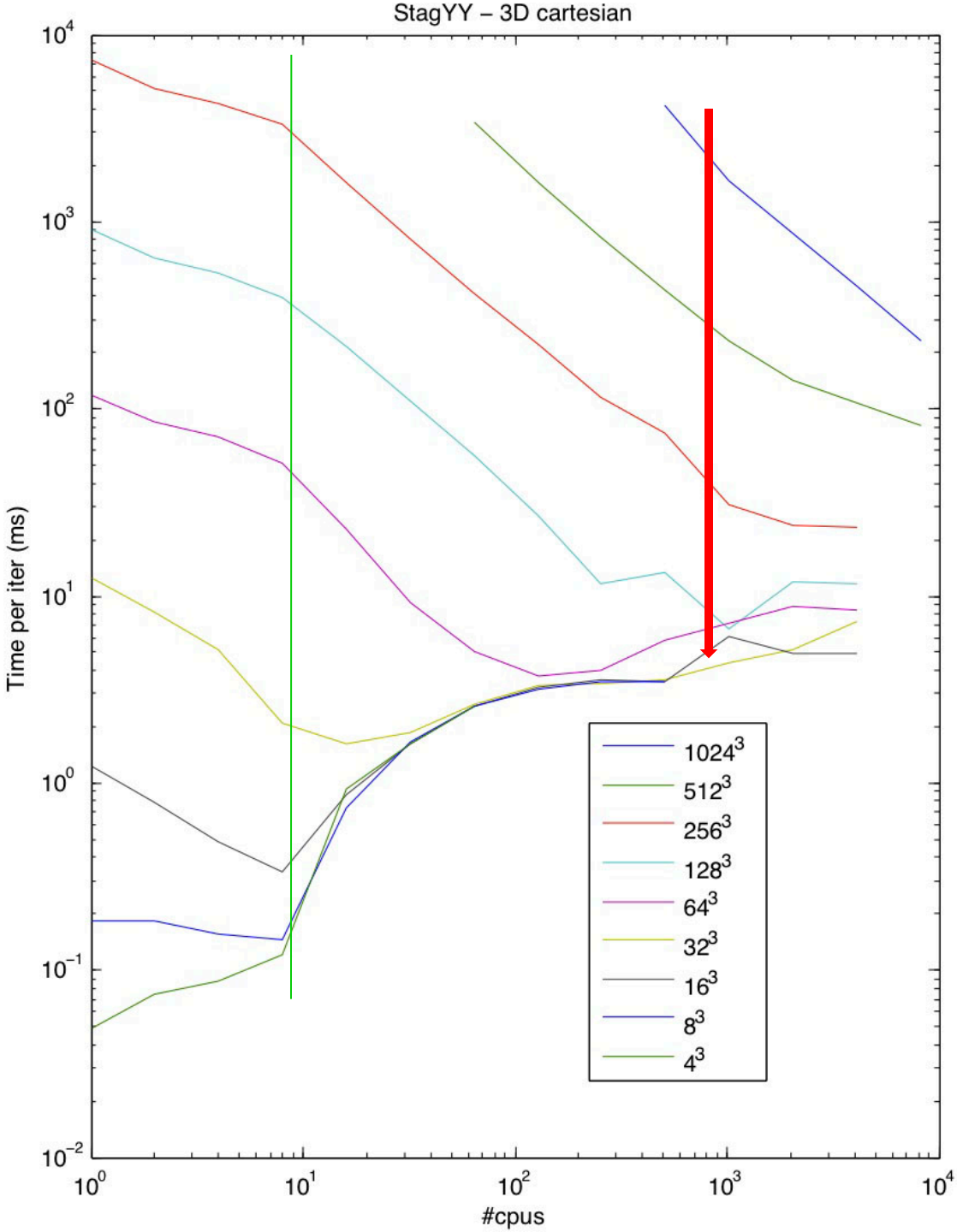
Application to StagYY
Cartesian or spherical



StagYY iterations: 3D Cartesian

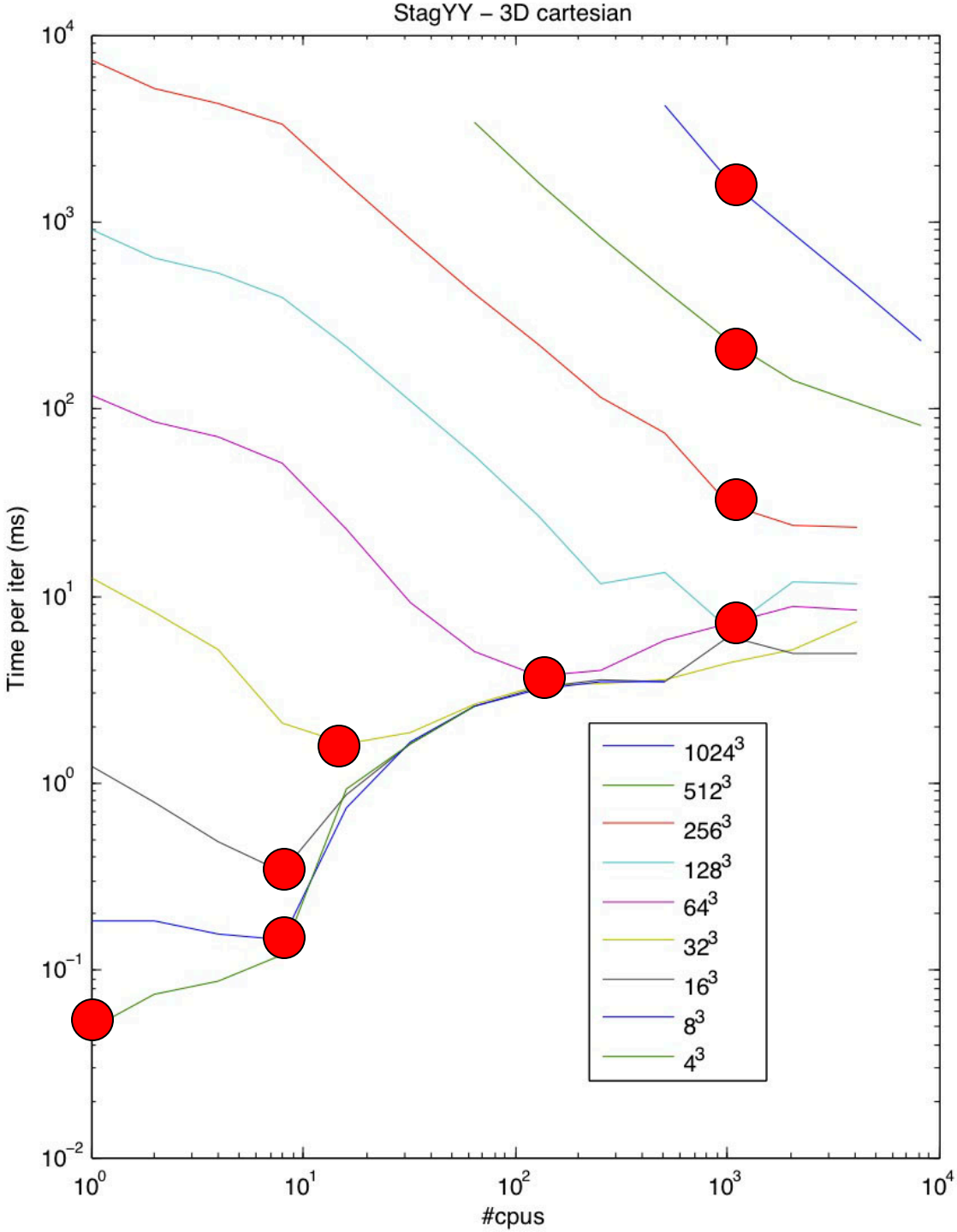
Change in scaling from same-node to cross-node communication

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

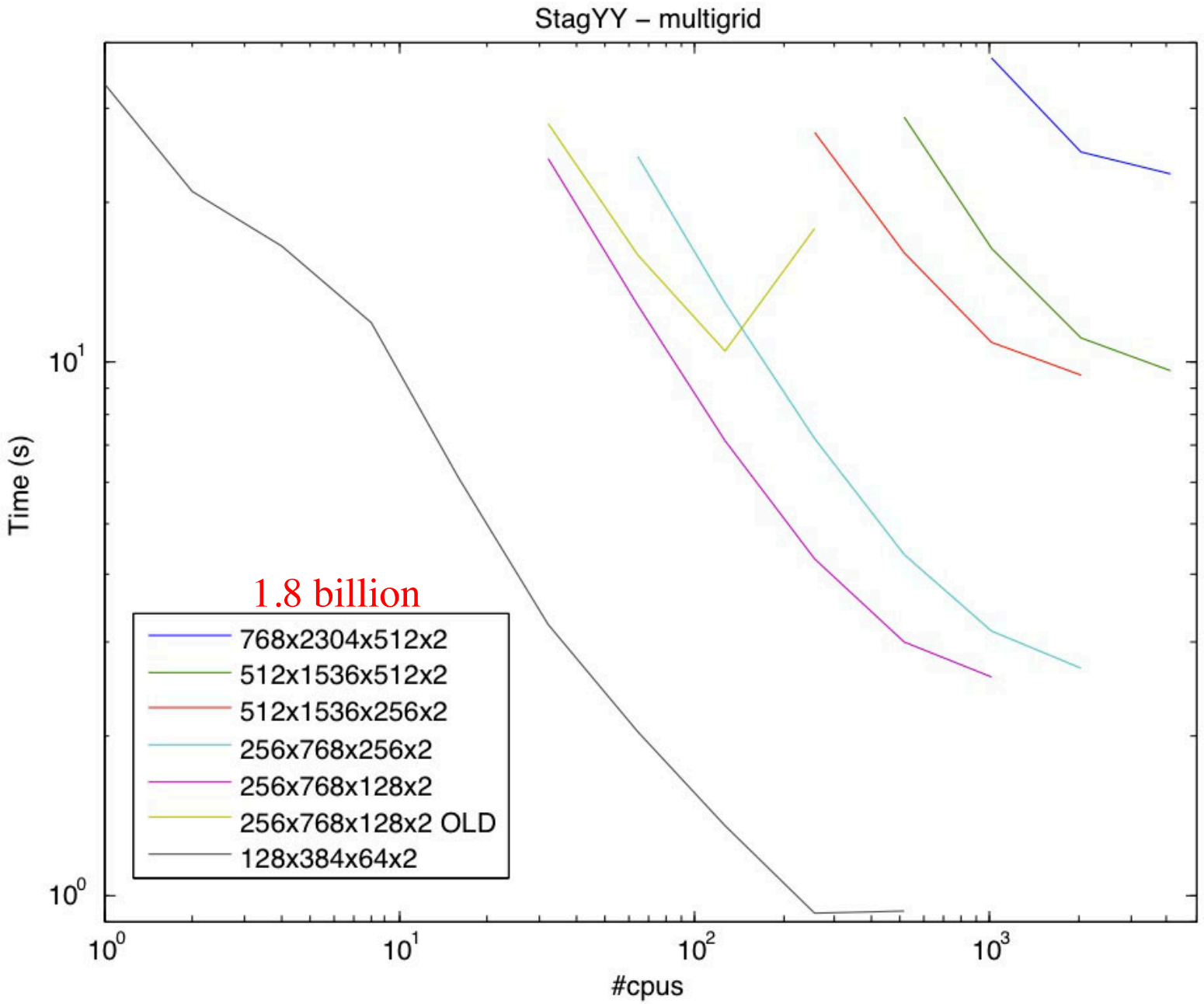


New treatment: follow minima

- 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 or CoArrays
- 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