Numerical Modelling in **FORTRAN** day 8

Paul Tackley, 2017
Today’s Goals

1. Introduction to parallel computing
   (applicable to Fortran or C; examples are in Fortran)

2. Finite Prandtl number convection
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 $\rightarrow$ 1.9 billion cells
  - $(2 \text{ km})^3$ cells $\rightarrow$ 123 billion cells
## Top 500 Supercomputers

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Supercomputing Center in Wuxi, China</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRPC</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>2</td>
<td>National Super Computer Center in Guangzhou, China</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>3</td>
<td>Swiss National Supercomputing Centre (CSCS), Switzerland</td>
<td>Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.66GHz, Aries interconnect, NVIDIA Tesla P100 Cray Inc.</td>
<td>361,760</td>
<td>19,590.0</td>
<td>25,326.3</td>
<td>2,272</td>
</tr>
<tr>
<td>4</td>
<td>Japan Agency for Marine-Earth Science and Technology, Japan</td>
<td>Gyoukou - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz ExaScaler</td>
<td>19,860,000</td>
<td>19,135.8</td>
<td>28,192.0</td>
<td>1,350</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Oak Ridge National Laboratory, United States</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>6</td>
<td>DOE/NNSA/LLNL, United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
<tr>
<td>7</td>
<td>DOE/NNSA/LANL/SNL, United States</td>
<td>Trinity - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.</td>
<td>979,968</td>
<td>14,137.3</td>
<td>43,902.6</td>
<td>3,844</td>
</tr>
<tr>
<td>8</td>
<td>DOE/SC/LBNL/NERSC, United States</td>
<td>Cori - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.</td>
<td>622,336</td>
<td>14,014.7</td>
<td>27,880.7</td>
<td>3,939</td>
</tr>
</tbody>
</table>
### Top 500 Supercomputer Sites

**Huge problems => huge computer**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Supercomputing Center in Wuxi, China</td>
<td><strong>Sunway TaihuLight</strong> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRPC</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>2</td>
<td>National Super Computer Center in Guangzhou, China</td>
<td><strong>Tianhe-2 (MilkyWay-2)</strong> - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>3</td>
<td>Swiss National Supercomputing Centre (CSCS) Switzerland</td>
<td><strong>Piz Daint</strong> - Cray XC50, Xeon E5-2690v3 12C 2.66GHz, Aries interconnect, NVIDIA Tesla P100 Cray Inc.</td>
<td>361,760</td>
<td>19,590.0</td>
<td>25,326.3</td>
<td>2.272</td>
</tr>
<tr>
<td>4</td>
<td>Japan Agency for Marine-Earth Science and Technology Japan</td>
<td><strong>Gyoukou</strong> - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz ExaScaler</td>
<td>19,860,000</td>
<td>19,135.8</td>
<td>28,192.0</td>
<td>1,350</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Oak Ridge National Laboratory United States</td>
<td><strong>Titan</strong> - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>6</td>
<td>DOE/NNSA/LLNL United States</td>
<td><strong>Sequoia</strong> - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
<tr>
<td>7</td>
<td>DOE/NNSA/LANL/SNL United States</td>
<td><strong>Trinity</strong> - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.</td>
<td>979,968</td>
<td>14,137.3</td>
<td>43,902.6</td>
<td>3,844</td>
</tr>
<tr>
<td>8</td>
<td>DOE/SC/LBNL/NERSC United States</td>
<td><strong>Cori</strong> - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.</td>
<td>622,336</td>
<td>14,014.7</td>
<td>27,880.7</td>
<td>3,939</td>
</tr>
</tbody>
</table>
Progress: iPhone – fastest computer in 1976 (cost: $8 million)

Then: the CRAY-1A had
- 12.5 nanosecond clock
- 8 vector registers (each vector register was 64 elements long)
- 1 million 64-bit words of high speed memory
- Executed over 80 megaflops (peak speed 160 megaflops)
- 4.5 times faster than the CDC 7600

Now: the iPhone 3G has
- 7.5 times more compute power
- 32 times more random access memory
- 3.33 times more flash storage

(photo taken at NCAR museum)
In Switzerland

<table>
<thead>
<tr>
<th>Rank</th>
<th>Center</th>
<th>System Description</th>
<th>Float Ops</th>
<th>FLOPS</th>
<th>Memory (GB)</th>
<th>Node Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Swiss National Supercomputing Centre (CSCS)</td>
<td>Piz Daint - Cray XC50, Xeon E5-2690v3</td>
<td>361,760</td>
<td>19,590.0</td>
<td>25,326.3</td>
<td>2,272</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 core 2.6GHz, Aries interconnect,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NVIDIA Tesla P100 Cray Inc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>Swiss National Supercomputing Centre (CSCS)</td>
<td>Piz Daint Multicore - Cray XC40, Xeon E5-2695v4</td>
<td>44,928</td>
<td>1,410.7</td>
<td>1,509.6</td>
<td>519</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18 core 2.1GHz, Aries interconnect Cray Inc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each node: 12-core Intel CPU + GPU

Each node: 2* 18-core Intel CPU
**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

1991: 512 CPUs, distributed memory

2010: 224,162 Cores, distributed + shared memory (12 cores per node)
Another possibility: build your 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.open-mpi.org/](http://www.open-mpi.org/)
How to parallelise a code: worked example
Example: Scalar Poisson eqn.

\[ \nabla^2 u = f \]

Finite-difference approximation:

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

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 \( R_{ij} \) 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

=ghost points
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
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 nx1, ny1, nz1 | #grid points in local subdomain
integer xmin,ymin,zmin | min. coordinate (1 if external bdry, 0 if internal)
integer :: nxz=1, nyz=1, nz1=1 | #cpus in each direction
integer myx, myy, myz | local subdomain coordinates
integer cpu_xm, cpu_xp, cpu_ym, cpu_ym, cpu_zm, cpu_zp | cpus holding adjacent subdomains
integer npx, ny, npx | #points along each side (to communicate)
integer ierr | needed for MPI routines in Fortran

! set up

call set_up_parallelisation()

allocate (u(-1:nx1,-1:ny1,-1:nz1), f(-1:nx1,-1:ny1,-1:nz1), r(-1:nx1,-1:ny1,-1:nz1))
u = 0.; f = 0.; r = 0.
forall (i=xmin:nx1-1, j=ymin:ny1-1, k=zmin:nz1-1) &
   f(i,j,k) = float(i+myx*nx1 + j+myy*ny1 + k+myz*nz1)/(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:nx1-1, j=ymin:ny1-1, k=zmin:nz1-1)
      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)
   end forall

   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 bdry, 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:nz), f(-1:nxl,-1:nyl,-1:nz), r(-1:nxl,-1:nyl,-1:nz))

u = 0.; f = 0.; r = 0.

forall (i=xmin:nxl-1, j=ymin:nyl-1, k=zmin:nz-1)
  f(i, j, k) = float(i*myx*nxl + j*myy*nyl + k*myz*nz)/(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: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) ) 
                 & + ( f(i, j, k) - alpha*r(i, j, k) )
enddo
integer ncpu, mycpu     ! #cpus and number of local cpu
integer nx1, nyl, nzl   ! grid points in local subdomain
integer xmin, ymin, zmin ! min. coordinate (1 if external bdry, 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, npx, npz   ! #points along each side (to communicate)
integer ierr, npx, npz   ! needed for MPI routines in Fortran

! set up

call set_up_parallelisation()

allocate (u(-1:nx1,-1:ny1,-1:nz1), f(-1:nx1,-1:ny1,-1:nz1), r(-1:nx1,-1:ny1,-1:nz1))
u = 0.; f = 0.; r = 0.

forall (i=xmin:nx1-1, j=ymin:ny1-1, k=zmin:nz1-1)    
  f(i,j,k) = float(i+myx*nx1 + j+myy*ny1 + k+myz*nz1)/(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:nx1-1, j=ymin:ny1-1, k=zmin:nz1-1)     &
    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)
    nmax = max(nxl,nyl,nzl)
    if(nx1==nmax) then
      ncx = ncx*2; nxl = nx1/2
    else if (ny1==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=(ny1+2)*(nz1+2) ! #points on each side to communicate
  npy=(nx1+2)*(nz1+2)
  npz=(nx1+2)*(ny1+2)
end subroutine set_up_parallelisation
Simplest communication

Not optimal – uses blocking send/receive
Better: using non-blocking (isend/irecv)

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

subroutine sidesld(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 sidesld

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 ($N_x^2$)

$\Rightarrow$ Communication/Computation $\propto 1/N_x$

$\Rightarrow$ 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: \( t = nL + bN^2 / B \)

\( (L = \text{Latency, } B = \text{bandwidth}) \)

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
⇒ 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
Time/iteration vs. #cpus

Quadrics, Gonzales-size cluster
Up to 2e5 CPUs (Quadrics communication)
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

1.8 billion
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/](https://computing.llnl.gov/tutorials/parallel_comp/)
Programming:
Finite Prandtl number convection (i.e., almost any fluid)

Ludwig Prandtl (1875-1953)
Values of the Prandt number $Pr$

$$Pr = \frac{\nu}{\kappa}$$

- Viscous diffusivity
- Thermal diffusivity

- Liquid metals: 0.004-0.03
- Air: 0.7
- Water: 1.7-12
- Rock: $\sim 10^{24}$ !!! (effectively infinite)
Finite-Prandtl number convection

- Existing code assumes infinite Prandtl number
  - also known as Stokes flow
  - appropriate for highly-viscous fluids like rock, honey etc.
- Fluids like water, air, liquid metal have a lower Prandtl number so equations must be modified
Applications for finite Pr

- Outer core (geodynamo)
- Atmosphere
- Ocean
- Anything that’s not solid like the mantle
Equations

- Conservation of mass (= ‘continuity’)
- Conservation of momentum (‘Navier-Stokes’ equation: $F=ma$ for a fluid)
- Conservation of energy

Claude Navier (1785-1836)
Sir George Stokes (1819-1903)
Finite Pr Equations

Navier-Stokes equation: $F=ma$ for a fluid

$$\rho \left( \frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} \right) = -\nabla P + \rho \nu \nabla^2 \vec{v} + 2\rho \vec{\Omega} \times \vec{v} + g \rho \alpha \hat{y} \rho \frac{\partial T}{\partial t} + \vec{v} \cdot \nabla T = \kappa \nabla^2 T + Q \nabla \cdot \vec{v} = 0$$

Validity for constant viscosity only

“ma”

$\rho=$ density, $\nu=$ kinematic viscosity, $g=$ gravity, $\alpha=$ thermal expansivity
Non-dimensionalise the equations

- Reduces the number of parameters
- Makes it easier to identify the dynamical regime
- Facilitates comparison of systems with different scales but similar dynamics (e.g., analogue laboratory experiments compared to core or mantle)
Non-dimensionalise to thermal diffusion scales

- Lengthscale $D$ (depth of domain)
- Temperature scale ($T$ drop over domain)
- Time to $D^2 / \kappa$
- Velocity to $\kappa / D$
- Stress to $\rho \nu \kappa / D^2$
Nondimensional equations

\[ \nabla \cdot \vec{v} = 0 \quad \frac{\partial T}{\partial t} + \vec{v} \cdot \nabla T = \nabla^2 T \]

\[ \frac{1}{\text{Pr}} \left( \frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} \right) = -\nabla P + \nabla^2 \vec{v} + \frac{1}{\text{Ek}} \tilde{\Omega} \times \vec{v} + \text{Ra} \, T \hat{y} \]

\[ \text{Pr} = \frac{\nu}{\kappa} \quad \text{Ek} = \frac{\nu}{2 \Omega D^2} \quad \text{Ra} = \frac{g \alpha \nabla T D^3}{\nu \kappa} \]

Prandtl number \hspace{1cm} \text{Ekman number} \hspace{1cm} \text{Rayleigh number}
As before, use streamfunction

\[ v_x = \frac{\partial \psi}{\partial y} \quad \quad v_y = -\frac{\partial \psi}{\partial x} \]

Also simplify by assuming \(1/\text{Ek}=0\)
Eliminating pressure

- Take curl of 2D momentum equation: curl of grad=0, so pressure disappears
- Replace velocity by vorticity: \( \vec{\omega} = \nabla \times \vec{v} \)
- in 2D only one component of vorticity is needed (the one perpendicular to the 2D plane), \( \nabla^2 \psi = \omega_z \)

\[
\frac{1}{Pr} \left( \frac{\partial \omega}{\partial t} + v_x \frac{\partial \omega}{\partial x} + v_y \frac{\partial \omega}{\partial y} \right) = \nabla^2 \omega - Ra \frac{\partial T}{\partial x}
\]
=> the streamfunction-vorticity formulation

\[
\frac{1}{\Pr} \left( \frac{\partial \omega}{\partial t} + v_x \frac{\partial \omega}{\partial x} + v_y \frac{\partial \omega}{\partial y} \right) = \nabla^2 \omega - Ra \frac{\partial T}{\partial x}
\]

\[
\nabla^2 \psi = -\omega \quad \left( v_x, v_y \right) = \left( \frac{\partial \psi}{\partial y}, -\frac{\partial \psi}{\partial x} \right)
\]

\[
\frac{\partial T}{\partial t} + \bar{v} \cdot \nabla T = \nabla^2 T + Q
\]
Note: Effect of high Pr

\[
\frac{1}{Pr} \left( \frac{\partial \omega}{\partial t} + v_x \frac{\partial \omega}{\partial x} + v_y \frac{\partial \omega}{\partial y} \right) = \nabla^2 \omega - Ra \frac{\partial T}{\partial x}
\]

If Pr->infinity, left-hand-side=>0 so equation becomes Poisson like before:

\[
\nabla^2 \omega = Ra \frac{\partial T}{\partial x}
\]
Taking a timestep

(i) Calculate $\psi$ from $\omega$ using: $\nabla^2 \psi = \omega$

(ii) Calculate $v$ from $\psi$:

$$\left( v_x, v_y \right) = \left( \frac{\partial \psi}{\partial y}, -\frac{\partial \psi}{\partial x} \right)$$

(iii) Time-step $\omega$ and $T$ using explicit finite differences:

$$\frac{\partial T}{\partial t} = -v_x \frac{\partial T}{\partial x} - v_y \frac{\partial T}{\partial y} + \nabla^2 T$$

$$\frac{\partial \omega}{\partial t} = -v_x \frac{\partial \omega}{\partial x} - v_y \frac{\partial \omega}{\partial y} + \Pr \nabla^2 \omega - Ra \Pr \frac{\partial T}{\partial x}$$
T time step is the same as before

\[
\frac{T_{\text{new}} - T_{\text{old}}}{\Delta t} = -v_x \frac{\partial T_{\text{old}}}{\partial x} - v_y \frac{\partial T_{\text{old}}}{\partial y} + \nabla^2 T_{\text{old}}
\]

\[
T_{\text{new}} = T_{\text{old}} + \Delta t \left( \nabla^2 T_{\text{old}} - v_x \frac{\partial T_{\text{old}}}{\partial x} - v_y \frac{\partial T_{\text{old}}}{\partial y} \right)
\]

\[\omega_{\text{new}} - \omega_{\text{old}} = \frac{-v_x \frac{\partial \omega_{\text{old}}}{\partial x} - v_y \frac{\partial \omega_{\text{old}}}{\partial y} + \text{Pr} \nabla^2 \omega_{\text{old}} - \text{Ra Pr} \frac{\partial T_{\text{old}}}{\partial x}}{\Delta t} \]

\[
\omega_{\text{new}} = \omega_{\text{old}} + \Delta t \left( \text{Pr} \nabla^2 \omega_{\text{old}} - v_x \frac{\partial \omega_{\text{old}}}{\partial x} - v_y \frac{\partial \omega_{\text{old}}}{\partial y} - \text{Ra Pr} \frac{\partial T_{\text{old}}}{\partial x} \right)
\]

w must now be time stepped in a similar way
START

INPUT
number grid points: nx, ny
Parameters: Ra, Pr
time-step as: a_dif, a_adv
integration time: total_time
Initial condition choice

TIME STEP
Calculate S (from w) -> v
Calculate dt_adv -> dt
Calculate del2(T), del2(w), dT/dx
Calculate v.grad(T) & v.grad(w)
T = T + dt*(d2T-vgradT)
w = w+dt*(Pr.d2w-vgradw+RaPrdTdx)
time = time + dt

INITIALISE
h=1/(ny-1)
dt_dif=a*h**2/max(1,Pr)
Allocate T, w, S, vx, vy
Initialise T & w fields
S=v=0

time>total_time?

END

No

Yes
Stability condition

Diffusion: \[ dt_{\text{diff}} = a_{\text{diff}} \frac{h^2}{\max(\text{Pr}, 1)} \]

Advection: \[ dt_{\text{adv}} = a_{\text{adv}} \min\left( \frac{h}{\max \text{val}(\text{abs}(vx))}, \frac{h}{\max \text{val}(\text{abs}(vy))} \right) \]

Combined: \[ dt = \min(dt_{\text{diff}}, dt_{\text{adv}}) \]
Modification of previous convection program

- Replace Poisson calculation of w with time-step, done at the same time as T time-step
- Get a compiling code!
- Make sure it is stable and convergent for values of Pr between 0.01 and 1e2
- Hand in your code, and your solutions to the test cases in the following slides
- **Due date: 18 December** (2 weeks from today)
Test cases

• All have nx=257, ny=65, Ra=1e5, total_time=0.1, and random initial T and w fields, unless otherwise stated

• Due to random start, results will not look exactly as these, but they should look similar (i.e., width of upwellings & downwellings & boundary layers similar, but number and placement of upwellings/downwellings different).
Pr=10
$Pr=1$
Pr=0.1
Pr=0.01, time=1.0
Pr=0.1, Ra=1e7