Outline

Parallel Algorithms for Partial Differential Equations

- Introduction
- Partial Differential Equations
- Finite Difference Time Domain
- Domain Decomposition
- Multi-block grids
- Particle tracking
- Finite Volumes
- Parallel-by-line algorithms: Compact FD and Spectral Methods
- Implicit Time algorithms and ADI
- Unstructured meshes
- Adaptive Mesh Refinement
- Master-slave approach
- A few references

Outline

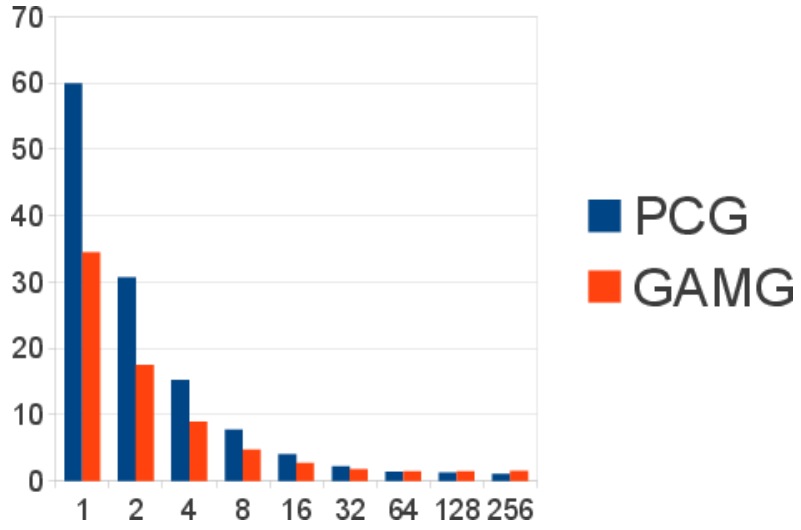
Parallel Algorithms

- We will talk about Parallel Algorithms for Scientific Computing
  - not from a theoretical point of view but a discussion of some typical "situations" you may encounter
  - focusing on Distributed Memory Layouts
  - addressing to possible C and Fortran MPI implementations
  - a few advanced MPI concepts will be discussed
  - we will show good practice, not always the best practice

- The purpose
  - giving ideas for setting up the (MPI) parallelization of your scientific code
  - understanding terminology and common techniques well implemented in the libraries you may want to use

Introduction

- The best serial algorithm is not always the best after parallelization (if the parallelization is possible at all!)
- Scalability analysis of a CFD RANS solver, simpleFoam (from the finite-volume *OpenFOAM* suite)
  - the most expensive computing section is the linear solver
  - time-step versus number of nodes - Blue Gene/Q architecture
  - PCG solver: preconditioned conjugate gradient solver
  - GAMG solver: generalised geometric-algebraic multi-grid solver



Introduction / 2

- Another basic concept about performances of parallel programming: the slowest rules!
  - the program ends when the slowest process finishes its work
  - if synchronizations are performed (**`MPI_Barrier`**), each process waits for the slowest process at each barrier, the result may be disastrous
- Beware of serial parts of the code usually performed by **`rank=0`** process (or by all processes)
  - remember Amhdal law and speed-up limit

$$1 + \frac{\text{Parallel Section Time}}{\text{Serial Section Time}}$$

  - critical especially for massively parallel applications (e.g., $N_{processes} > 100$)

Introduction / 3

- Basic principle: each process should perform the same amount of work
  - if each process performs the same computations (Single Program Multiple Data paradigm) the main task is to split the data among processes
- Communication issues need to be considered
  - minimizing: prefer decomposition where MPI exchanges are small
  - balancing: include communication time when estimating the process work
  - optimizing: use efficiently the MPI procedures
    - non-blocking communications
    - topologies
    - Remote Memory Access (RMA)
    - patterns
    - ...

Introduction / 4

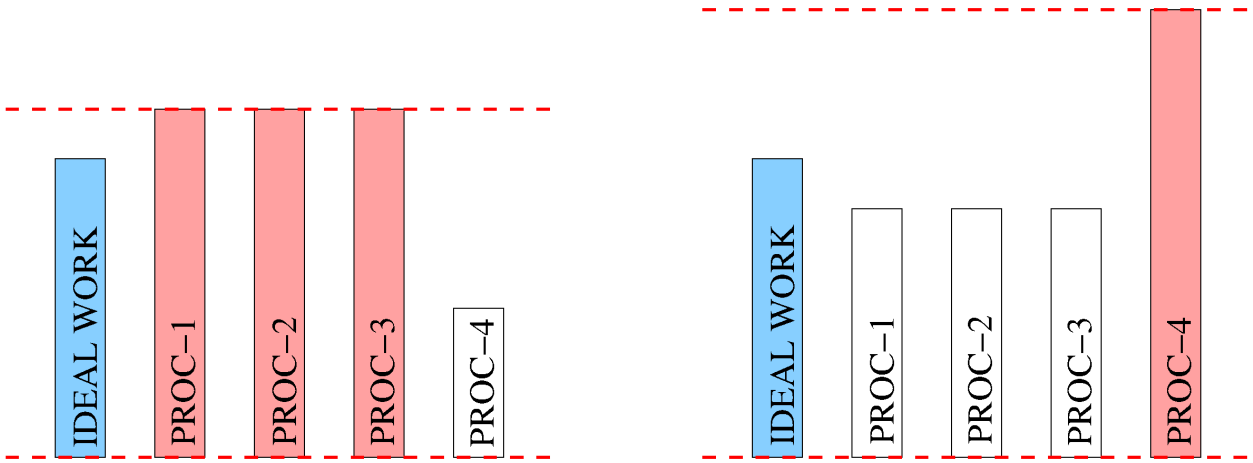
- If W_T is the total work split among N processes, $P = 1...N$, a global unbalancing factor may be evaluated as

$$\max_{P=1,N} \left| \frac{W_T - N \cdot W_P}{W_T} \right|$$

- In the pure SPMD case, the amount of work may be roughly substituted with the amount of processed data
  - grid-points, volumes, cells, Fourier modes, particles,...
- But including the communication cost may be a good idea especially when communication time is not negligible

Introduction / 5

- Beware: unbalancing is not a symmetrical concept
  - N-1 processes slow, one process fast: it is ok
  - N-1 processes fast, one process slow: catastrophic! Unfortunately, it may happen for the notorious **`rank=0`**



Programming models

- Mixing Distributed and Shared memory programming models (Hybrid programming, e.g. MPI+OpenMP) may help:
  - to allow for parallelization up to a larger number of cores
  - to reduce communication times
  - not a panacea, MPI does not perform real communications when the data are in the same node
  - refer to the next lessons about that
- Heterogeneous computing is today more and more on the rise
  - depending on the device, the code need to be significantly modified (Nvidia GPU)
  - or at least massive scalability must be ensured (Intel MIC)
  - how to efficiently decompose work among host and devices having different potentiality?

Outline

Partial Differential Equations

- Consider a set of Partial Differential Equations

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(\mathbf{u})$$

  - where **u** is a vector function of space **x** and time t
  - **f** is the forcing term involving time, space and derivatives

$$\frac{\partial^\alpha \mathbf{u}}{\partial \mathbf{x}^\alpha}$$

- Depending on PDE features (**f**, BC, IC, ...), many algorithms may be used to numerically solve the equation, e.g.
  - finite difference
  - finite volumes
  - finite elements
  - spectral methods

Outline

Finite Difference Time Domain

- Example: 1d convection-diffusion equation

$$\frac{du}{dt} = c\frac{du}{dx} + \nu\frac{d^2u}{dx^2}$$

- Uniform discretization grid

$$x_i = (i-1)\cdot dx \qquad ; \qquad i = 1, N$$

- Explicit Time advancement (e.g. Euler)

$$\frac{u_i^{(n+1)} - u_i^{(n)}}{dt} = c\left(\frac{du}{dx}\right)_i^{(n)} + \nu\left(\frac{d^2u}{dx^2}\right)_i^{(n)}$$

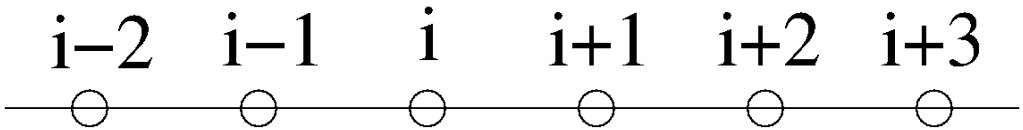
- Need to evaluate derivatives on grid points

Finite Difference Time Domain

- Using Explicit Finite differences, the derivatives are approximated by linear combination of values in the “stencil” around node *i*

$$\left(\frac{d^{\alpha}u}{dx^{\alpha}}\right)_i = \sum_{k=-l,r} a_k \, u_{i+k}$$

- Coefficients a_k are chosen to optimize the order of accuracy, the harmonic behaviour,...
- The stencil may be symmetric or not depending on
  - the needed numerical properties (e.g. upwind schemes)
  - boundary treatment



Finite Difference Time Domain

- Example: 4-th order centered FD:

$$\left(\frac{du}{dx}\right)_i = \frac{1/12u_{i-2} - 2/3u_{i-1} + 2/3u_{i+1} - 1/12u_{i+2}}{dx}$$

- For points close to boundaries two approaches are common
  - use adequate asymmetric stencil (hopefully preserving the numerical properties), e.g.

$$\left(\frac{du}{dx}\right)_1 = \frac{-25/12u_1 + 4u_2 - 3u_3 + 4/3u_4 - 1/4u_5}{dx}$$

$$\left(\frac{du}{dx}\right)_2 = \ldots$$

  - use halo (ghost) regions to maintain the same internal scheme

Outline

FDTD: Distributing work

- Distributing work is rather simple because the computations are mainly “local” (FDTD explicit in time and explicit in space)
- “global” array is an abstraction: there is no global array allocated anywhere
  - beware: there should be no global array allocated anywhere
- For 1d cases, each process stores arrays with size N/N_{PROC}
  - if N is not multiple of N_{PROC}, you have to deal with the remainders
  - distribute over $N_{PROC} - 1$ processes and assign the remainder to the proc N_{PROC}
  - split across N_{PROC} processes and assign the remainder $r < N_{PROC}$ one per process (usually the last r ranks are selected, expecting that $rank = 0$ could be already a bit overloaded)

FDTD: Distributing work

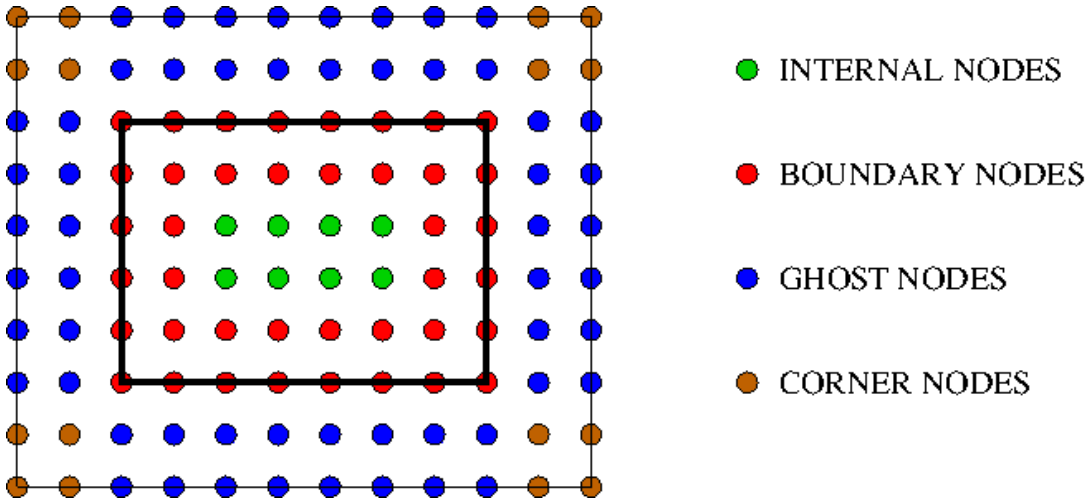
- The index of the array is a local index, the global index may be easily rebuilt
  - considering the remainder is zero

```
i_glob = i+rank*n
```

- Dynamic memory allocation is needed to avoid recompilation when changing the number of processes
- In Fortran, you can preserve the global indexing by exploiting the user-defined array indexing **`u(istart:iend)`**

FDTD: Distributing work

- Terminology
  - internal points: evolved points not depending on points belonging to other domains
  - boundary points: evolved points depending on points belonging to other domains
  - halo points: points belonging to another domain such that there is a boundary point which depends on them
  - corner points: just a geometrical description for Cartesian grids, may be halo points or useless points depending on the algorithm

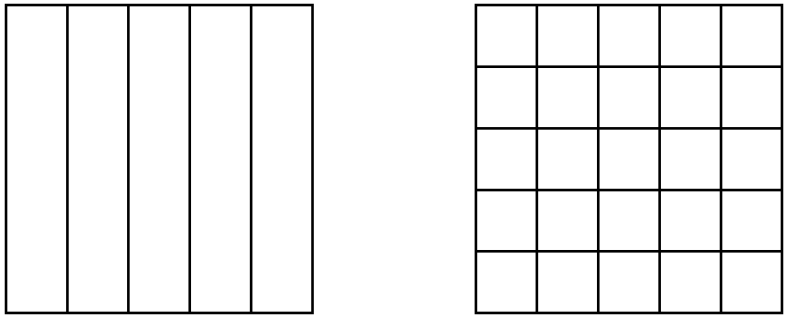


FDTD: Distributing work

- In order to calculate the next state of variables on points, some data from adjacent processes are needed
  - need to communicate these regions at (least at) each time-step: **halo exchange**
- Artificial boundaries are created for each process
  - BC must be imposed only on original boundaries, artificial boundaries need to exchange data
- To perform FD derivatives the halo choice allows to preserve results
  - anyhow, to decrease communications, asymmetric stencils may be adopted for small terms of the equations
- Ghost values need to be updated whenever a derivative is calculated, e.g. to compute $\frac{\partial}{\partial x}\left(u\frac{\partial u}{\partial x}\right)$
  - ghost updating of u to calculate $\frac{\partial u}{\partial x}$
  - ghost updating of $\frac{\partial u}{\partial x}$ to compute the final result

1D versus 2D/3D decomposition

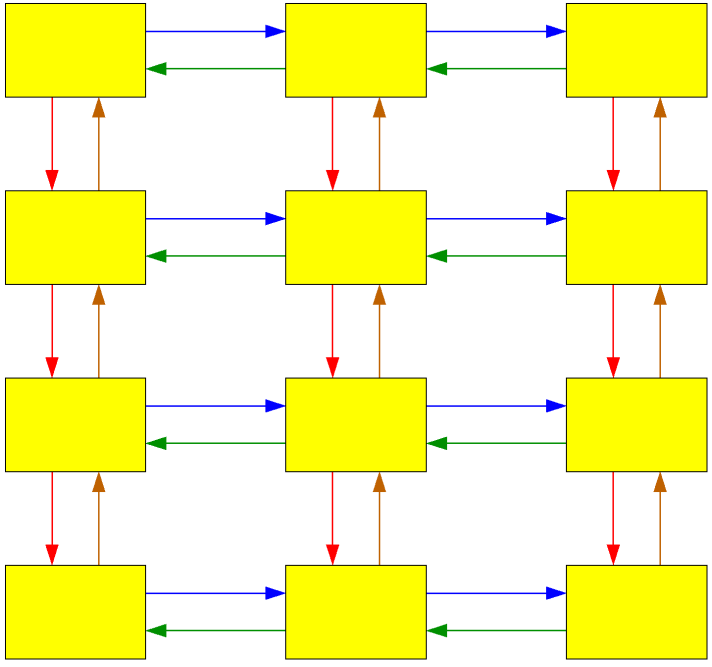
- Consider Cartesian domain
- 2D-Decomposition increases the number of processes to communicate with
- But may reduce the amount of communications
  - 1D-Decomposition: each process sends and receives $2N$ data
  - 2D-Decomposition: each process sends and receives $4N/\sqrt{N_{PROC}}$ data
- 2D-Decomposition is convenient for massively parallel cases



- Same idea for 3D cases: 3D decomposition may scale up to thousands of processes

Halo exchange

- Map of Halo exchange for a 2D grid



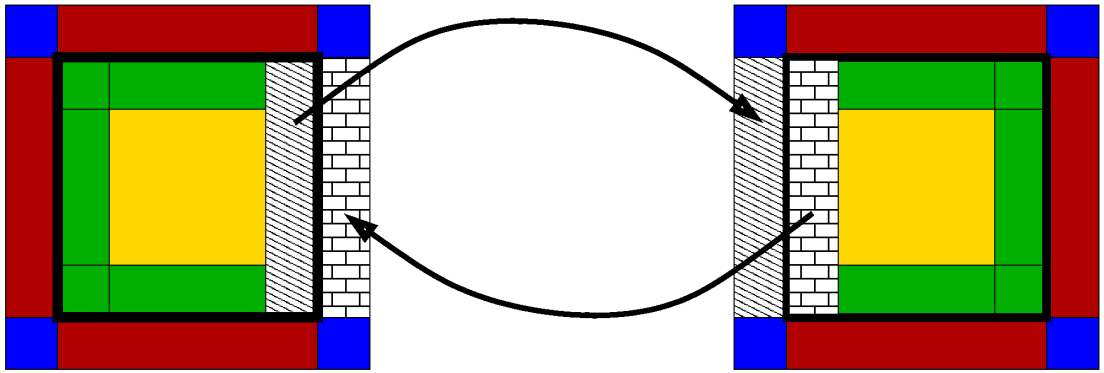
- From the process point of view, a very asymmetric configuration!
  - if periodic boundaries are included, symmetry may be recovered

Cartesian Communicators

- Best practice: duplicate communicator to ensure communications will never conflict
  - required if the code is made into a component for use in other codes
- But there is much more: MPI provides facilities to handle such Cartesian Topologies
- Cartesian Communicators
  - **`MPI_Cart_create`**, **`MPI_Cart_Shift`**. **`MPI_Cart_Coords`**,...
  - activating reordering, the MPI implementation may associate processes to perform a better process placement
  - communicating trough sub-communicators (e.g., rows communicator, columns communicator) may improve performances
  - useful also to simplify coding

Clarifying halo exchange

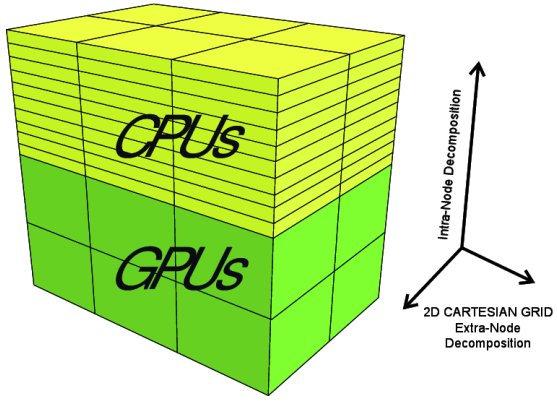
- Let us clarify: boundary nodes are sent to neighbour processes to fill their halo regions



- Remember: MPI common usage is two sided (the only way until MPI 2)
  - if process A sends data to process B, both A and B must be aware of it and call an MPI routine doing the right job
  - one-sided communications (RMA) were introduced in MPI 2, very useful but probably not crucial for the basic domain decomposition

Heterogeneous decomposition

- Assume your architecture features nodes with 8 cores and 2 GPUs each
  - for your code GPU is R_{GPU} times faster than a single core
  - but you do not want to waste the power of CPUs
- You have to devise a non-uniform decomposition, e.g.



- Take care of the possible unbalancing: a small relative unbalancing for a core may be dramatic wrt GPU performance degradation
  - give to CPUs less work than than theoretical optimal values

Heterogeneous decomposition / 2

- Of course, you need to write a code running two different paths according to its rank
- A naive but effective approach: set **NCOREXNODE** and **NGPUXNODE**

```
GPU = .false.
if(mod(n_rank,NCOREXNODE) .lt. NGPUXNODE) then
  call acc_set_device(acc_device_nvidia)
  call acc_set_device_num(mod(n_rank,NCOREXNODE),acc_device_nvidia)
  print*,'n_rank: ',n_rank,' tries to set GPU: ',mod(n_rank,NCOREXNODE)
  my_device = acc_get_device_num(acc_device_nvidia)
  print*,'n_rank: ',n_rank,' is using device: ',my_device
  print*,'Set GPU to true for rank: ',n_rank
  GPU = .true.
endif
........
if(GPU) then; call var_k_eval_acc(ik) ;     else; call var_k_eval_omp(ik) ;     endif
if(GPU) then; call update_var_k_mpi_acc() ; else; call update_var_k_mpi_omp() ; endif
if(GPU) then; call bc_var_k_acc() ;         else; call bc_var_k_omp() ;         endif
if(GPU) then; call rhs_k_eval_acc() ;       else; call rhs_k_eval_omp() ;       endif
........
```

- use `MPI_Comm_split` to be more robust

Pattern SendRecv

- The basic pattern is based on **`MPI_Sendrecv`**
  - e.g.: send to left and receive from right, and let MPI handling the circular dependencies
  - by the way, **`MPI_Sendrecv`** is commonly implemented by **`MPI_Isend`**, **`MPI_Irecv`** and a pair of **`MPI_Wait`**
  - beware: send to left and receive from left cannot work, why?
- For 2d decomposition, at least 4 calls are needed
  - send to left and receive from right
  - send to right and receive from left
  - send to top and receive from bottom
  - send to bottom and receive from top
- 4 calls more if you need corners
  - send LU and receive from RB
  - ...

Non-blocking communications

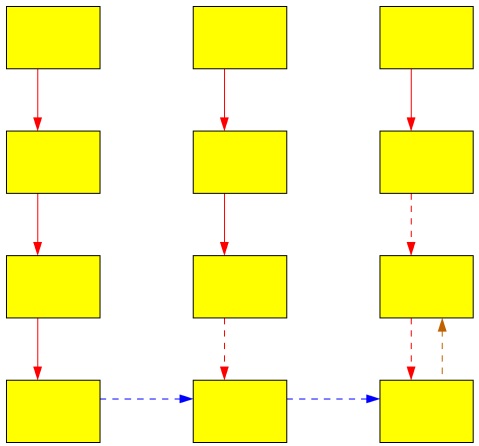
- Non-blocking functions may improve performances
  - reducing the artificial synchronization points
  - but the final performances have to be tested (and compared to the **`MPI_Sendrecv`** ones)
- A possible choice (consider neighbours ordered as down,right,up,left)

```
do i=1,n_neighbours
   call MPI_Irecv(....)
enddo
do i=1,n_neighbours
   call MPI_Send(....)
enddo
call MPI_Waitall(....)  ! wait receive non-blocking calls
```

  - Does not perform well in practice. Why?

Non-blocking communications / 2

- Contention at the receiver: the bandwidth is shared among all processes sending to the same process
  - receive do not interfere with Sends
- Sends block and control timing
  - Ordering of sends introduces delays
  - Bandwidth is being wasted
- How many steps are needed to finish the exchange?
  - this is the first one (and the answer is 6)



Non-blocking communications / 3

- Prefer the pattern

```
do i=1,n_neighbours
   call MPI_Irecv(....)
enddo
do i=1,n_neighbours
   call MPI_ISend(....)
enddo
call MPI_Waitall(....)  ! wait non-blocking send and receive
```

  - This may end in 4 steps (the minimal theoretical limit)
- Actual performances depend on architecture, MPI implementation,...
- Manually controlling the scheduling is a possibility (e.g., Phased Communication) but consider it only if the current exchange times are huge

Using buffers

- Halo data are usually not contiguous in memory
  - naive approach: use buffers to prepare data to send or receive
  - when copying back received data be careful about copying only actually received buffers (not corresponding to physical boundary conditions)

```
for(j = 1; j<=mymsize_y; j++) buffer_s_rl[j-1] = T[stride_y+j];
for(j = 1; j<=mymsize_y; j++) buffer_s_lr[j-1] = T[mymsize_x*stride_y+j];
for(i = 1; i<=mymsize_x; i++) buffer_s_tb[i-1] = T[stride_y*i+1];
for(i = 1; i<=mymsize_x; i++) buffer_s_bt[i-1] = T[stride_y*i+mymsize_y];

MPI_Sendrecv(buffer_s_rl, mymsize_y, MPI_DOUBLE, dest_rl, tag,
             buffer_r_rl, mymsize_y, MPI_DOUBLE, source_rl, tag,
             cartesianComm, &status);
MPI_Sendrecv(buffer_s_lr, mymsize_y, MPI_DOUBLE, dest_lr, tag+1,
               buffer_r_lr, mymsize_y, MPI_DOUBLE, source_lr, tag+1,
               cartesianComm, &status);
MPI_Sendrecv(buffer_s_tb, mymsize_x, MPI_DOUBLE, dest_tb, tag+2,
               buffer_r_tb, mymsize_x, MPI_DOUBLE, source_tb, tag+2,
               cartesianComm, &status);
MPI_Sendrecv(buffer_s_bt, mymsize_x, MPI_DOUBLE, dest_bt, tag+3,
               buffer_r_bt, mymsize_x, MPI_DOUBLE, source_bt, tag+3,
               cartesianComm, &status);

if(source_rl>=0)for(j=1;j<=mymsize_y;j++)T[stride_y*(mymsize_x+1)+j]=buffer_r_rl[j-1];
if(source_lr>=0)for(j=1;j<=mymsize_y;j++)T[j]=buffer_r_lr[j-1];
if(source_tb>=0)for(i=1;i<=mymsize_x;i++)T[stride_y*i+mymsize_y+1]=buffer_r_tb[i-1];
if(source_bt>=0)for(i=1;i<=mymsize_x;i++)T[stride_y*i]=buffer_r_bt[i-1];
```

Fortran alternative

- Using Fortran, buffers may be automatically managed by the language
  - unlike C counter-parts, Fortran pointers may point to non-contiguous memory regions

```
buffer_s_rl => T(1,1:mymsize_y)
buffer_r_rl => T(mymsize_x+1:1:mymsize_y)
call MPI_Sendrecv(buffer_s_rl, mymsize_y, MPI_DOUBLE_PRECISION, dest_rl, tag, &
     buffer_r_rl, mymsize_y, MPI_DOUBLE_PRECISION, source_rl, tag, &
     cartesianComm, status, ierr)
```

- Or you can trust the array syntax
  - probably the compiler will create the buffers
  - and in some cases, it may fail, why?

```
call MPI_Sendrecv(T(1,1:mymsize_y), mymsize_y, MPI_DOUBLE_PRECISION, dest_rl, tag, &
    T(mymsize_x+1:1:mymsize_y), mymsize_y, MPI_DOUBLE_PRECISION, source_rl, tag, &
    cartesianComm, status, ierr)
```

Using Data-types

- It it possible to avoid the usage of buffers?
  - in principle yes, data-types are a solution
  - type **`vector`** is enough for halo regions of Cartesian grids
  - or use **`subarray`** which is more intuitive
  - perform MPI communications sending a MPI vector or subarray as a single element
- Performances actually depend on the underlying implementation
- Again, try it if you see that buffering times are significant

Hiding communications

- It is possible to devise MPI communication patterns capable of minimizing the communication times
- First idea: use non-blocking send/recv and perform the part of algorithm which does need halo values before waiting for the communication completion
  1. start non-blocking send
  2. start non-blocking receive
  3. advances internal grid points (halo values are not needed)
  4. wait for send/recv completion (probably finished when arriving here)
  5. advances boundary grid points (halo values are needed now)
- Good idea but, unfortunately the advancement algorithm need to be split
  - hard work for complex codes
- Anything else?

Hiding communications / 2

- Using exchange buffers instead of directly exchanging the evolved variables may be exploited
- 1st iteration: start non-blocking receive
- Halo updating
  - fill send buffer
  - start non-blocking send
  - waitall receive
  - copy from receive buffers
  - start non-blocking receive
  - waitall send
- Advance from 1st to 2nd iteration
- Halo updating
- Advance from 2st to 3nd iteration
- . . . . . it works!

Hiding communications / 3

- Before starting optimizing the patterns, check with a profiler the actual impact of communications in your code
- From a real-world example:
  - code for Direct Numerical Simulation of turbulence
  - explicit in time and space, 3D Cartesian decomposition
  - weak scaling up to 32768 cores with efficiency around 95% using blocking **`MPI_Sendrecv`** and buffers!
  - strong scaling is much less efficient: which problem are you addressing?
- Does it worth while optimizing it?
- Dealing with heterogeneous computing, hiding communications may require additional effort
  - try to hide not only MPI communication costs, but also host/device communications
  - patterns may be tricky and still dependent on programming paradigm (CUDA, OpenCL,. . .)

Collective and Reductions

- Even in explicit algorithms, exchanging halos is not enough to carry on the computation
  - often, you need to perform “reductions”, requiring collective communications
- Consider you want to check the behaviour of “residuals” (norm of field difference among two consecutive time-steps)
  - **`MPI_Allreduce`** will help you
- The situation is more critical for implicit algorithms
  - the impact of collective communications may the actual bottle-neck of the whole code (see later)

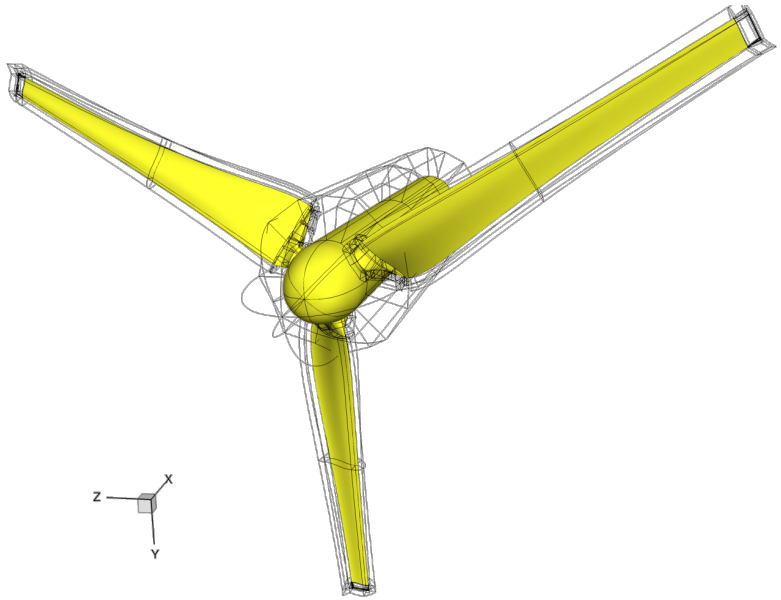
Collective and Reductions / 2

- Up to MPI-2, collective communications were always blocking
  - to perform non-blocking collectives you had to use threads (Hybrid Programming)
- Using MPI-3 non-blocking collective procedures are available
  - check if your MPI implementation supports MPI-3
  - anyhow, the usage of threads may be still a good option for other reasons (again, study Hybrid Programming)
- But the problem is that, often, collective operations must be executed and finished before going on with the computation
  - select carefully the algorithm to implement

Outline

Multi-block structured grids

- Compared to single-block structured grids, an improvement in order to deal with complex geometries
  - especially when different geometrical parts need a different treatment, i.e. different equations, e.g. fluid-structure interaction
  - structured grids may be quite easily generated with current grid generators
  - capability of dealing with moving and overlapping grids (CHIMERA)



Multi-block and OO

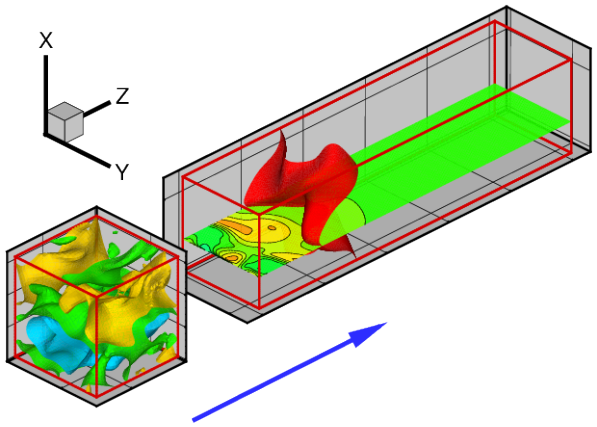
- Multi-block may be well implemented in Object Oriented Programming, i.e. a block may a be an object

```
type, public:: Type_Block
   integer(I4P)::                Ni=0,Nj=0,Nk=0
   integer(I4P)::                N_var
contains
  procedure:: init           => init_block
  procedure:: advance        => advance_block
  procedure:: halo_exchange  => halo_exchange_block
  procedure:: print          => print_block
endtype Type_Block
```

- And, if needed, organized using lists, trees, hash-tables, . . .
- Plan the code design accurately before starting

Multi-block: Load Balancing

- As always, load balancing is crucial
- First case: just a few of large blocks
  - e.g., two blocks where the first one is used to generate the inlet Boundary Conditions for the second one



- Simplest strategy: decompose each block using all MPI processes
- For each time-step each process sequentially evolves both blocks

Multi-block: Load Balancing / 2

- Two-fold MPI communications: intra-node and extra-node
  - non-blocking MPI routines are needed for extra-node communications

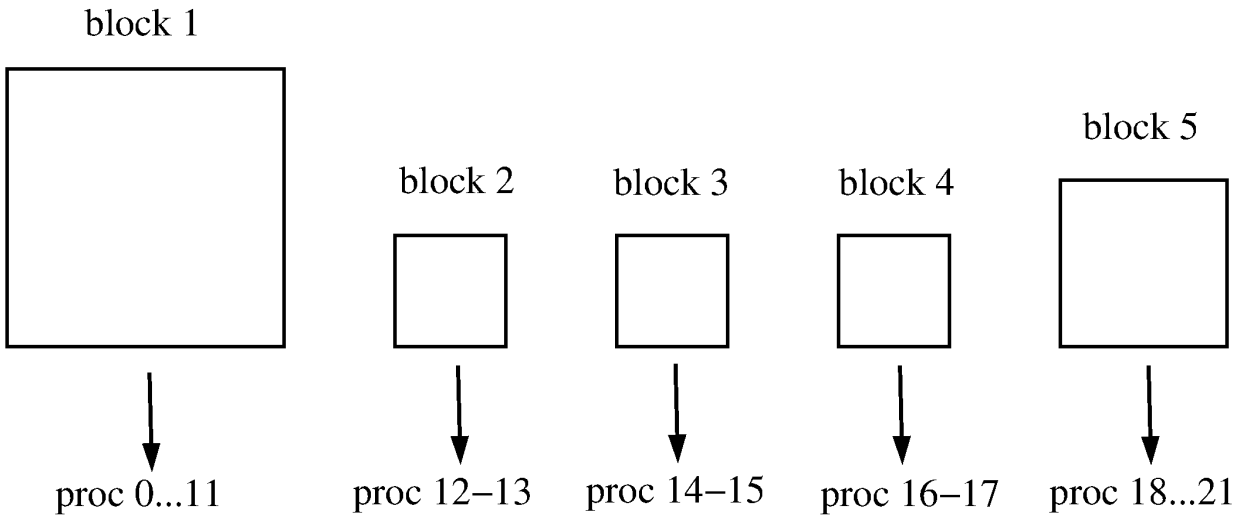
```
do it=1,itmax
   ! First block:
   call update_var_mpi_acc_001()  ! MPI: blocking update intra-block halo
   call exc_bc_var_acc_001()      ! MPI: call non-blocking extra-block halo

   ! Second block:
   call update_var_mpi_acc_002()  ! MPI: blocking update intra-block halo
   call exc_bc_var_acc_002()      ! MPI: call non-blocking extra-block halo

   ! Wait extra-block exchange var
   call exc_wait_var_acc_001()    ! MPI: wait to complete extra-block halo
   call exc_wait_var_acc_002()    ! MPI: wait to complete extra-block halo

   ! First block: step II
   call bc_var_acc_001()          ! impose boundary conditions
   call rhs_eval_acc_001()        ! compute forcing terms
   call var_eval_acc_001(ik)      ! advance solution

   ! Second block: step II
   call bc_var_acc_002()          ! impose boundary conditions
   call rhs_eval_acc_002()        ! compute forcing terms
   call var_eval_acc_002(ik)      ! advance solution
enddo
```

Multi-block: Load Balancing / 3

- Second case
  - the number of blocks is a bit larger
  - some blocks are too small to be split among all the processors
- Use Multiple Instruction Multiple Data approach: group processes and assign groups to blocks
  - the simplest approach is to give a weight W_I to each block depending on the work-load per point
  - and to assign processes to block I having number of points N_I proportionally to its work-load

$$N_{P,I} = \frac{W_I \cdot N_I}{\sum W_J \cdot N_J} N_P$$

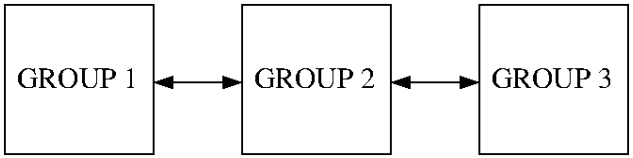
Multi-block: Load Balancing / 4



- Manually handling intra-node and extra-node communications may become a nightmare
  - split processes using MPI communicators and use intra-communicator usual domain decomposition
  - and **`MPI_COMM_WORLD`** or MPI Inter-Communicators to exchange data between different groups of processes

Multi-block: Load Balancing / 5

- **`MPI_Comm_split`**: split processes in groups
- **`MPI_Intercomm_create`**: create intercommunicators among different groups



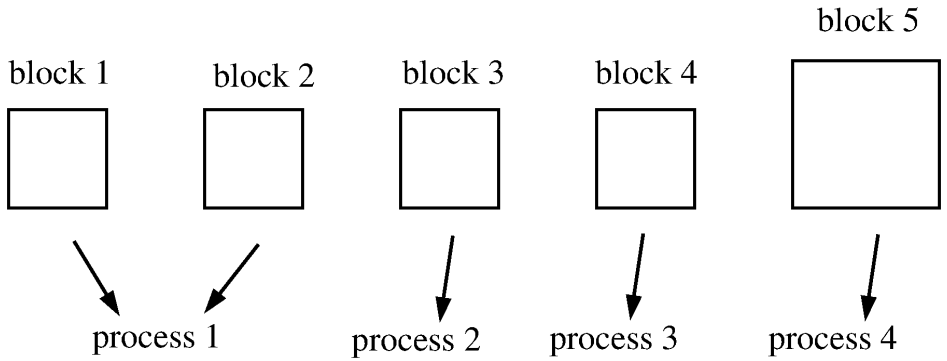

```
/* User code must generate membershipKey in the range [0, 1, 2] */
membershipKey = rank % 3;
/* Build intra-communicator for local sub-group */
MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
/* Build inter-communicators.  Tags are hard-coded. */
if (membershipKey == 0)  /* Group 0 communicates with group 1. */
{ MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1, 1, &myFirstComm); }
else if (membershipKey == 1)  /* Group 1 communicates with groups 0 and 2. */
{ MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0, 1, &myFirstComm);
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2, 12, &mySecondComm); }
else if (membershipKey == 2) /* Group 2 communicates with group 1. */
{ MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1, 12, &myFirstComm); }
/* Do work ... */
/* Free communicators... */
```

Multi-block: Load Balancing / 5

- Third case
  - the number of blocks is large
  - and the sizes may be different
- Common strategy
  - avoid intra-block MPI parallelization
  - if possible, use a shared-memory (OpenMP) intra-node parallelization
- Group blocks and assign groups to processes
  - obviously, the number of blocks must be greater or equal than the number of processes
  - in any case, to ensure a proper load balancing an algorithm has to be devised

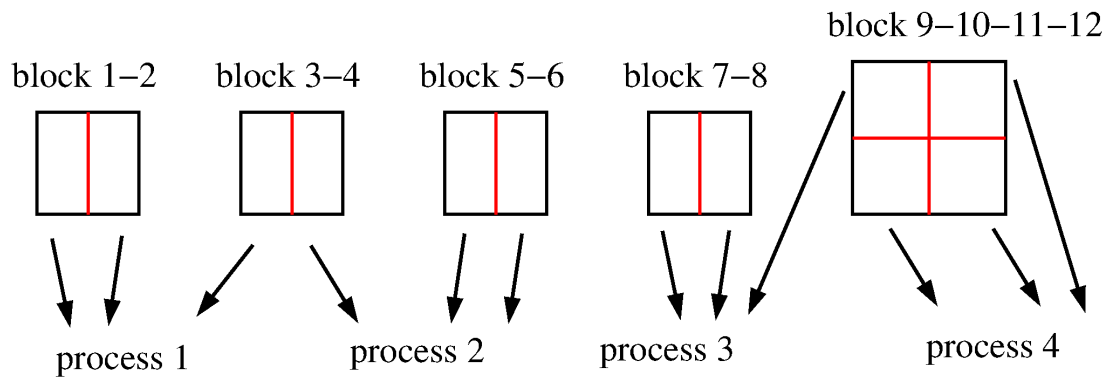
Multi-block: Load Balancing / 6

- Load Balancing Naive Algorithm
  - sort the block in descending order according to their work-loads
  - assign each block to a process until each process has one block
  - assign each of the remaining blocks to the most unloaded block
- Consider an unlucky case
  - 4 blocks having 1 million points each and 1 block having 2 million points
  - the best strategy results into the very unbalanced distribution
    `(1m+1m)-2m-1m-1m`



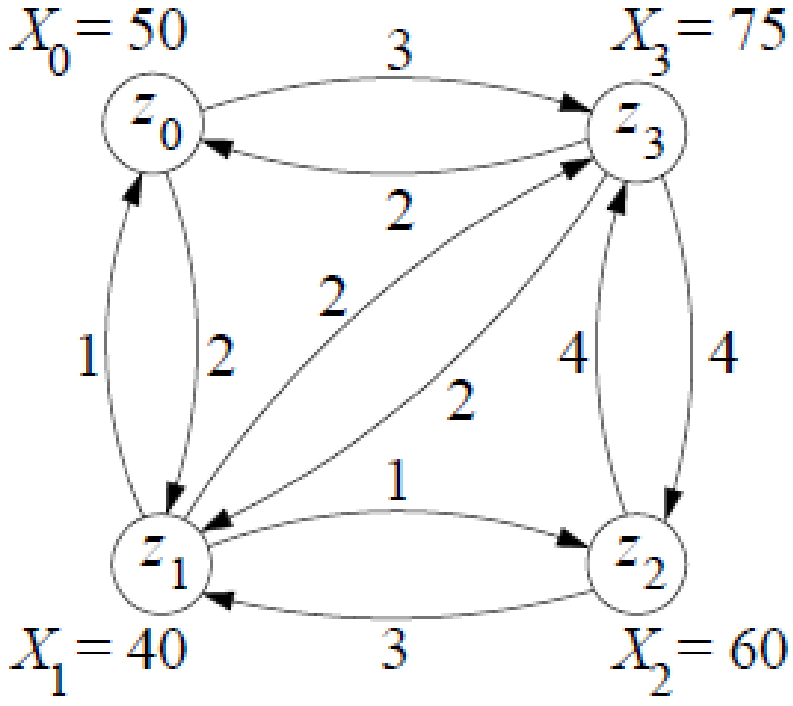
Multi-block: Load Balancing / 7

- Improve algorithm allowing block splitting
  - iterative algorithm: at each iteration sort and assign blocks to less loaded processes and check if the unbalancing factor is less than the goal tolerance
  - if not, split the largest block of the most unbalanced process along the largest direction and restart the algorithm until the required balancing is achieved
  - consider possible constraints: e.g., if using multi-grid schemes you require that each block has enough power of 2 after block splitting
  - devising a robust algorithm is not trivial



Multi-block: Load Balancing / 8

- Can you do better?
  - include communication costs when estimating work load
  - especially significant when dealing with overlapping grids
  - use a graph representation with weighted edges



Multi-block: Exchanging halo data

- Considering the “many blocks per process” configuration
  - before starting, store the blocks and process owners to communicate with (send and/or receive)
  - at each time step exchange data using adequate patterns
  - no simple sendrecv structure may be used (sender and receivers are not symmetrically distributed)
- A possible pattern
  1. copy from arrays to send buffers
  2. non-blocking recv from all relevant processes to buffers
  3. non-blocking send to all relevant processes to buffers
  4. loop over messages to receive
     - 4.a waitany catches the first arrived message
     - 4.b copy received buffers to variables
  5. waitall send

Multi-block: Exchanging halo data


```
! 1.  copy from arrays to send buffers
! ............
! 2.  non-blocking recv from all relevant processes to buffers
  do i_id=1,maxrecvid_num
     recvid = myrecvid(i_id)
     call receiveblockdata_fromid(receivearray(1,1,i_id),recvid,  &
         numrecv(i_id)*sendrecv_datasize,receiverequests(i_id))
  enddo
! 3.  non-blocking send to all relevant processes to buffers
  do i_id=1,maxsendid_num
     sendid = mysendid(i_id)
     call sendblockdata_toid(sendarray(1,1,i_id),sendid,          &
         numsend(i_id)*sendrecv_datasize,sendrequests(i_id))
  enddo
! 4.  loop over messages to receive
  do i_id=1,maxrecvid_num
     4.a waitany catches the first arrived message
     call waitanymessages(receiverequests,maxrecvid_num,reqnum)
     4.b copy received buffers to variables
     do i_in_id=1,numrecv(reqnum)
        a = receiveindices(i_in_id,1,reqnum)
        b = receiveindices(i_in_id,2,reqnum)
        iq1 = receiveindices(i_in_id,3,reqnum)
        imax1 = receiveindices(i_in_id,4,reqnum)
        jmax1 = receiveindices(i_in_id,5,reqnum)
        call copyqreceivedata(imax1,jmax1,a,b,q(iq1),receivearray,i_in_id,reqnum)
     enddo
  enddo
! 5.  waitall send
  call waitallmessages(sendrequests,maxsendid_num)
```

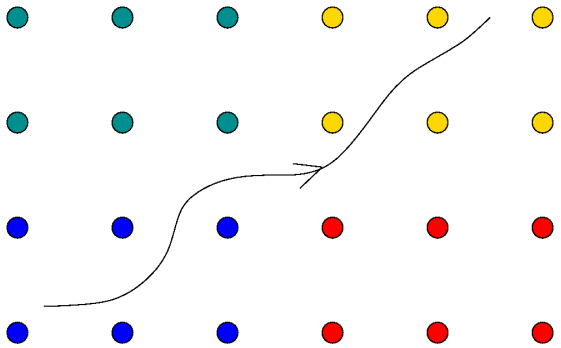
Outline

Particle tracking

- Consider you want to track particles moving according to the values of the velocity on your grid-points
  - e.g., evolve Eulerian flow field advancing Navier-Stokes equation
  - and simulate pollutant dispersion evolving Lagrangian particle paths
- If velocity values are known at grid points, to get the value of velocity of the particle you have to interpolate from surrounding points
- Beware: we are only going to deal with non-interacting particles
  - particle dynamics may be much much richer, we are not discussing molecular dynamics here
  - and the issues arising in different contexts may be much different and complex

Particle and processes

- Considering multi-dimensional decompositions, it is clear that particle tracking is one of the case for which corner data are required



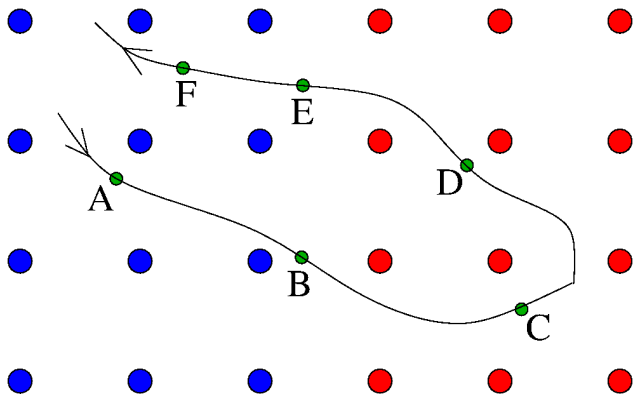
- When considering domain decomposition, at least three issues must be considered with care
  - load balancing including particles cost
  - changes in processes owning particles
  - dynamic memory layouts for particle storing

Particle and processes / 2

- Particle data need to be communicated from one process to another
  - reasonings about blocking/non-blocking communications and MPI patterns still apply
- If the cost of particle computing is high, the symmetry of Cartesian load balancing could be not enough anymore
  - in the simplest cases, symmetrically assigning a different amount of points to processes may solve the problem (see heterogeneous decomposition example)
  - in the worst cases, e.g., when particle clustering occurs, no simple symmetry is still available and the Cartesian Communicator is not the right choice
  - graph topology? multi-block? unstructured grid?

Particle and processes / 3

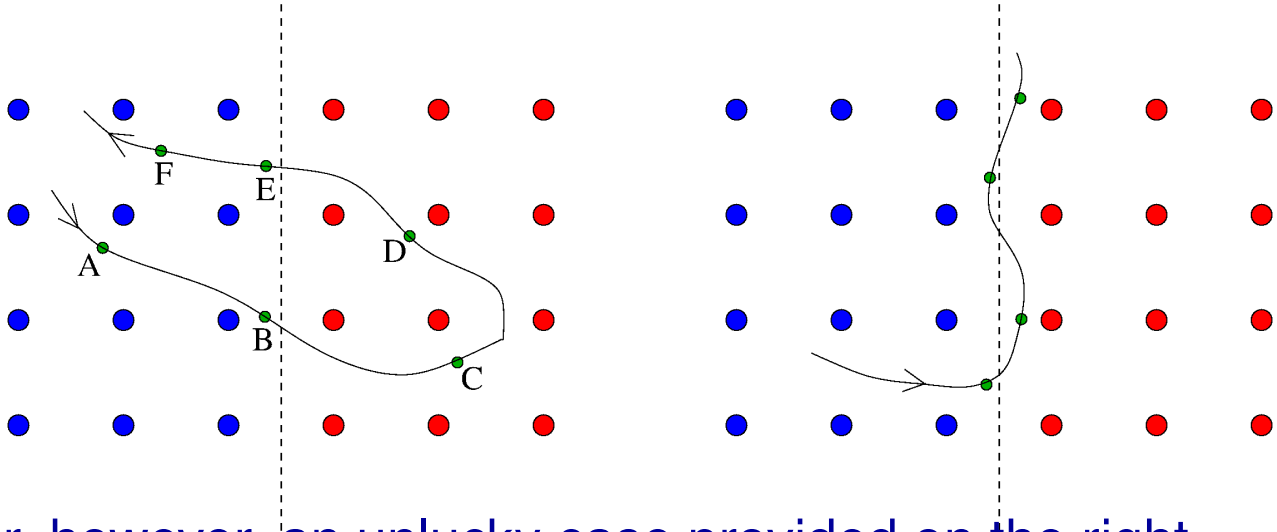
- Consider a simple domain decomposition with 2 processes (1st=blue, 2nd=red)



- Try to follow a particle from A to F positions
  - the process owning this particle has to change to access velocity field values
  - it is clear that process 1 should own the particle when passing trough A and F, while process 2 owns the particle for C and D positions

Particle and processes / 4

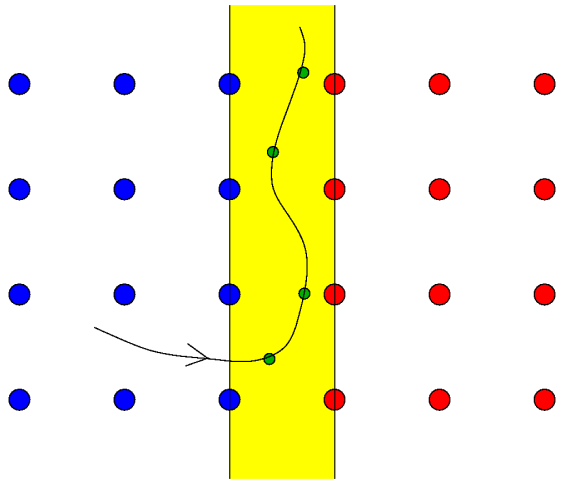
- What about B and E positions?
  - the first idea is to consider an artificial line in the middle of the process boundaries and assign particles wrt this edge
  - with this approach, in the sketched case both B and E would belong to the first process



- Consider, however, an unlucky case provided on the right
  - many particle communications would be needed
  - is it possible to devise something better?

Particle and processes / 5

- Conceive a dynamical domain decomposition
  - the region among “left” and “right” points does not statically belongs to a process
- Particle coming from the left still belong to left process as long as they are in the inter-region
  - the same for the right side
- With this approach, even in the unlucky case, the amount of communications is small



Particle and processes / 6

- A non trivial problem is devising a memory structure able to host particle data migrating from one process to another one
  - the problem: the particles to be exchanged are not known a priori

```
typedef struct particle {
  double pos [3]; double mass;
  int type;
  int number; // char name [80];
} particle;
```

- A linked list is a common solution
  - deleting an element from a list is easy moving pointers
  - include a name or a number tagging a particle to follow it when moving across processes

```
struct particle_list {
   particle p;
   struct particle_list * next;
};
```

Particle and processes / 7

- Using lists may be not efficient as you like when performing loops
  - or you are more familiar with arrays and you do not want to change

```
particle p[max_loc_particles];
```

- The array must be able to host new particles coming from other processes
  - add a field to particle struct (or use a special tag, e.g. 0) for empty places
  - work only on active “particles”
- To efficiently fill that places, you need to define and update
  - the number of free places for each process
  - an auxiliary array pointing to the empty places

```
int n_free_places;
particle free_places[max_loc_particles];
```

Outline

Parallel-by-point algorithms

- Explicit space/time FD are “parallel-by-point” algorithms
  - the computations can be done at each grid point independently of the computations at the other grid points
- explicit FD is parallel-by-point:

$$u_i \Rightarrow \left(\frac{\partial u}{\partial x}\right)_i$$

```
do i=istart,iend
   du_dx(i) = 1./(2.*dx)*(u(i+1)-i(i-1))
enddo
```

- explicit time advancement is parallel-by-point:

$$u_i^{(n)} \Rightarrow u_i^{(n+1)} = c\left(\frac{du}{dx}\right)_i^{(n)} + \nu\left(\frac{d^2u}{dx^2}\right)_i^{(n)}$$

- By the way, these algorithms allow easy shared-memory parallelizations
  - core vectorization, using SSE or AVX units
  - multi-core thread parallelization (pThread or OpenMP)

Finite Volumes

- A conservation law problem

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \mathbf{f}(\mathbf{u}) = q$$

may be discretized by integrating over each cell i of the mesh

$$\frac{d\mathbf{u}_i}{dt} + \frac{1}{V_i}\int_{S_i} \mathbf{f}(\mathbf{u}) \cdot \mathbf{n} dS = \frac{1}{V_i}\int_{V_i} q dV$$

  - $\mathbf{u}_i$ stands for the mean value

$$\mathbf{u}_i = \frac{1}{V_i}\int_{V_i} u dV$$

- Compared to FD, one of the main advantages is the possibility to handle unstructured grids
- Vertex-centered or Cell-centered (or mixed) configurations exist

Finite Volumes / 2

- To obtain a linear system, integrals must be expressed in terms of mean values
- For Volume integrals, midpoint rule is the basic option

$$\mathbf{q_i} = \frac{1}{V_i}\int_{V_i} q dV \simeq q(\mathbf{x}_i)$$

- For Surface integrals

$$\frac{1}{V_i}\sum_k \int_{S_{i,k}} \mathbf{f}(\mathbf{u})\cdot\mathbf{n_k} dS$$

interpolation is needed to obtain the functions values at quadrature points (face value f_f) starting from the values at computational nodes (cell values f_P and f_N)

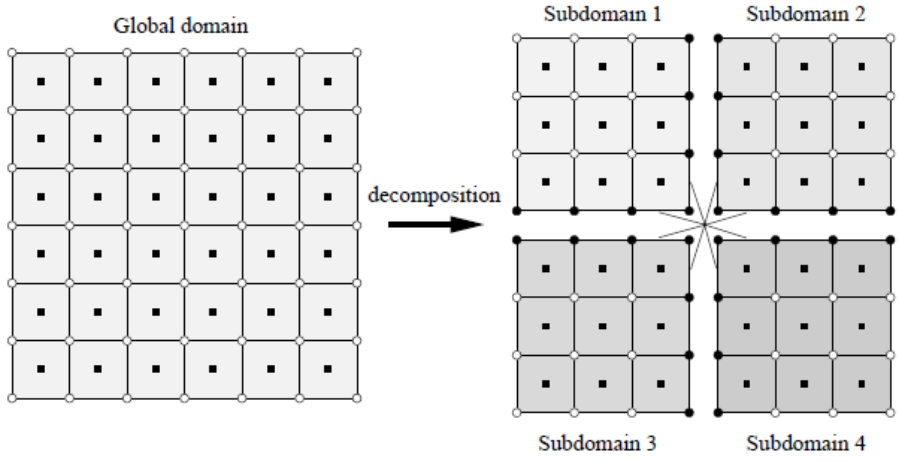
$$f_f = D\cdot f_P + (1-D)\cdot f_N$$

Finite Volumes / 3

- Finite Volume space discretization is parallel-by-point (but often time advancement is not)
- Interpolation is the most critical point wrt parallelism
- Consider a simple FVM domain decomposition: each cell belongs exactly to one processor
  - no inherent overlap for computational points
- Mesh faces can be grouped as follows
  - Internal faces, within a single processor mesh
  - Boundary faces
  - Inter-processor boundary faces: faces used to be internal but are now separate and represented on 2 CPUs. No face may belong to more than 2 sub-domains

Finite Volumes / 4

- The challenge it to efficiently implement the treatment of inter-processor boundaries



- Note: domain decomposition not trivial for complex (unstructured) geometries to achieve load balancing
- Two common choices
  - Halo Layer approach
  - Zero Halo Layer approach

FV: Halo Layer approach

- Considering

$$f_f = D \cdot f_P + (1 - D) \cdot f_N$$

  in parallel, f_P and f_N may live on different processors
- Traditionally, FVM parallelization uses halo layer approach (similar to FD approach): data for cells next to a processor boundary is duplicated
- Halo layer covers all processor boundaries and is explicitly updated through parallel communications calls
- Pro: Communications pattern is prescribed, only halo information is exchanged
- Con: Major impact on code design, all cell and face loops need to recognise and handle the presence of halo layer

FV: Zero Halo Layer approach

- Use out-of-core addressing to update boundaries
- Object-Oriented programming is of a great help (virtual functions)
- Assuming f_P is local, f_N can be fetched trough communication
- Note that all processors perform identical duties: thus, for a processor boundary between domain A and B, evaluation of face values can be done in 3 steps:
  1. Collect a subset internal cell values from local domain and send the values to the neighbouring processor
  2. Receive neighbour values from neighbouring processor
  3. Evaluate local processor face value using interpolation
- Pro: Processor boundary update encapsulates communication to do evaluation: no impact in the rest of the code
- Con: requires strong knowledge about OO, and, what about performance?

Outline

Parallel Algorithms for Partial Differential Equations

- Introduction
- Partial Differential Equations
- Finite Difference Time Domain
- Domain Decomposition
- Multi-block grids
- Particle tracking
- Finite Volumes
- Parallel-by-line algorithms: Compact FD and Spectral Methods
- Implicit Time algorithms and ADI
- Unstructured meshes
- Adaptive Mesh Refinement
- Master-slave approach
- A few references

Compact Finite Differences

- Is the space discretization always parallel-by-point? No, consider "implicit finite differences" usually called "compact"
- Explicit FD:
  - a single derivative value depends on values over a stencil

$$\left(\frac{d^\alpha u}{dx^\alpha}\right)_i = \sum_{k=-l,r} a_k \, u_{i+k}$$

  - matrix form

$$\frac{d^\alpha u}{dx^\alpha} = A\,u$$

- Compact FD:
  - the derivative values over a stencil depends on values over a (possibly different) stencil

$$\sum_{K=-L,R} \left(\frac{d^\alpha u}{dx^\alpha}\right)_{i+K} = \sum_{k=-l,r} a_k \, u_{i+k}$$

  - matrix form

$$B\,\frac{d^\alpha u}{dx^\alpha} = A\,u \Rightarrow \frac{d^\alpha u}{dx^\alpha} = B^{-1}\,A\,u$$

Compact Finite Differences

- Matrices A and B are banded (tridiagonal, pentadiagonal,...)
- Thomas algorithm is the best serial choice to invert such matrices

$$a\,x_{i-1} + b_i\,x_i + c_i\,x_{i+1} = d_i$$

- Forward and backward substitution are the core of the algorithm
  - forward sweep ($i = 1, ..., n$)

$$c'_i = \frac{c_i}{b_i - c'_{i-1}a_i} \quad ; \quad d'_i = \frac{d_i - d'_{i-1}a_i}{b_i - c'_{i-1}a_i}$$

  - back substitution ($i = n-1, .., 1$)

$$x_n = d'_n \quad ; \quad x_i = d'_i - c'_i\,x_{i+1}$$

- The order of loops is crucial since each iteration depends on the previous one: how to handle parallelization?

Compact Finite Differences / 2

- Considering a 3D problem, use a 2D domain decomposition
- When deriving along one direction, e.g. *x*-direction, transpose data so that the decomposition acts on the other two directions, e.g. *y* and *z*
- For each *y* and *z*, the entire *x* derivatives may be evaluated in parallel by Thomas algorithm
- Compact FD is an example of “parallel-by-line” algorithm
- Transpose data, i.e. **`MPI_Alltoall`**, has a (significant) cost:
  - may be slow
  - is only “out of place”, beware of memory usage
- Other possibilities?
  - use another algorithm instead of Thomas one: e.g., cyclic reduction may be better parallelized
  - compare the performances with the Transpose+Thomas choice

Spectral Methods

- Another class of methods, based on (Fast) Fourier Transform
  - may be very accurate
  - some equations get strongly simplified with this approach
- Considering 3D problems, a 3D-FFT is performed sequentially transforming *x*, *y* and *z* direction
- Since FFT usually employs a serial algorithm, a 3D-FFT is another example of “parallel-by-line” algorithm
- Transposition of data is the common way to handle parallelization of FFT
- Study carefully your FFT library
  - FFTW is the widespread library, also providing MPI facilities and a specialized MPI transpose routine capable of handling in-place data
  - often vendors FFTs perform better

Outline

Time advancement

- Explicit Time advancement algorithms are widely used
  - multi-stage (e.g. Runge-Kutta)
  - multi-step (e.g. leap-frog)
  - Lax-Wendroff
  - ...
- ...and are the best choice wrt parallelization
- However, implicit algorithms may be preferable for several reasons, e.g. to enlarge stability limits and achieve a faster convergence of steady-state problems
  - "implicit" means that Right Hand Side has to be evaluated using the "new" time
  - e.g., Poisson equation with Crank-Nicolson method (unconditionally stable)

$$\frac{u^{(n+1)} - u^{(n)}}{Dt} = \frac{1}{2}\left[\left(\frac{\partial^2 u}{\partial x^2}\right)^{(n+1)} + \left(\frac{\partial^2 u}{\partial y^2}\right)^{(n+1)} + \left(\frac{\partial^2 u}{\partial x^2}\right)^{(n)} + \left(\frac{\partial^2 u}{\partial y^2}\right)^{(n)}\right]$$

Time advancement

- Adopting the matrix form

$$A\,u^{(n+1)} = B\,u^{(n)}$$

  it results that a linear system has to be solved
- Thomas algorithm is not applicable because the bands of matrix are not contiguous
- The direct solution is costly, while an efficient approximate solutions may be obtained using iterative methods, e.g. conjugate gradient method
- Anyhow, the shape of A may be simple, how to exploit it?

ADI

- Some numerical schemes strongly simplify the parallelization
- Alternating direction implicit method (ADI)

$$\frac{u^{(n+1/2)} - u^{(n)}}{Dt} = 0.5 \left[\left(\frac{\partial^2 u}{\partial x^2} \right)^{(n+1/2)} + \left(\frac{\partial^2 u}{\partial y^2} \right)^{(n)} \right]$$

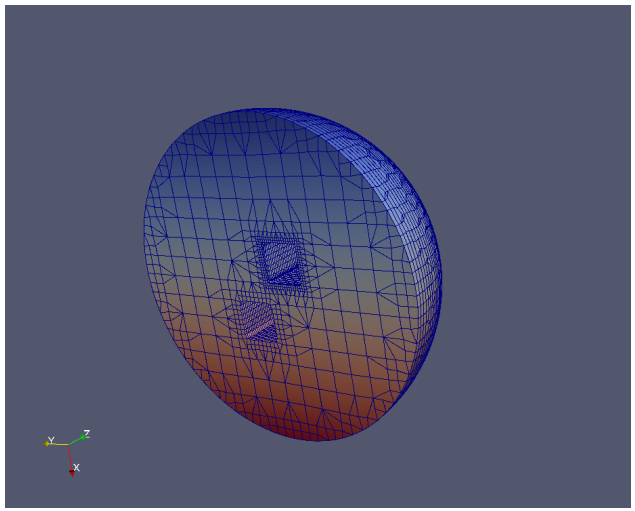
$$\frac{u^{(n+1)} - u^{(n+1/2)}}{Dt} = 0.5 \left[\left(\frac{\partial^2 u}{\partial x^2} \right)^{(n+1/2)} + \left(\frac{\partial^2 u}{\partial y^2} \right)^{(n+1)} \right]$$

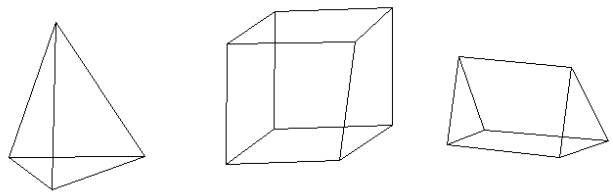
- The system is symmetric and tridiagonal and may be solved using Thomas algorithm
  - handling parallelization is not difficult transposing data

Outline

Unstructured meshes

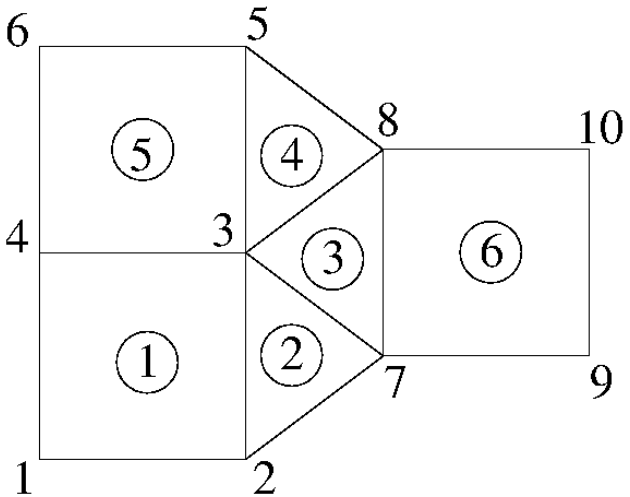
- Probably the most widespread strategy to handle complex geometries
  - but not the only one, e.g. AMR, immersed-boundary, . . .
- Idea: discretize the computational domain using polyhedron cells
  - in 3D, each cell has vertexes, edges, faces
  - according to the algorithm (FVM, FEM,...) you have to handle variables located on different zones of the cells
  - cells are usually tetrahedrons, hexahedrons or prisms

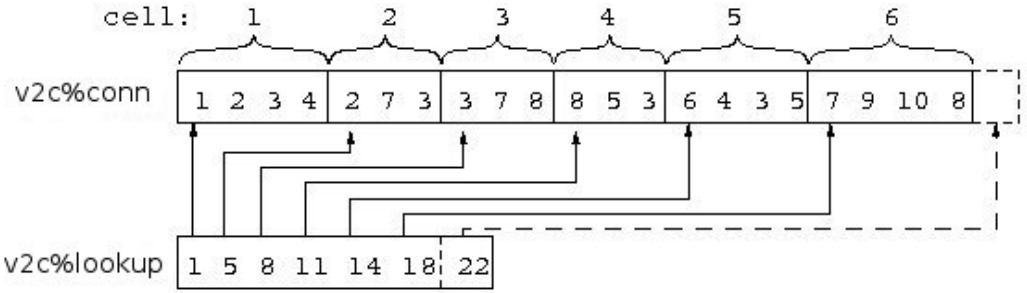




UM: CSR

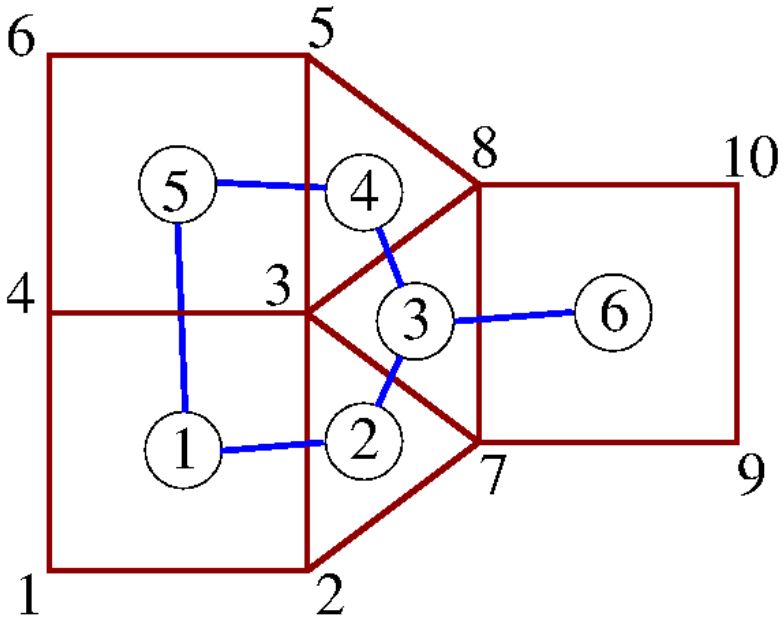
- When parallelizing codes running on unstructured meshes, an important issue is to decompose cells between processes
  - when dealing with huge meshes, mesh creation should be performed in parallel, too (e.g., snappyHexMesh tool provided by OpenFOAM)
- Let us detail how to describe the mesh topology: Compressed Sparse Row Format





UM: decomposition

- From the connectivity topological description, it is possible to build the dual graph based on cell centers or on cell vertexes
- To describe the dual graph, it is possible to list all the adjacent cells (or vertexes) for each cell (or vertex)

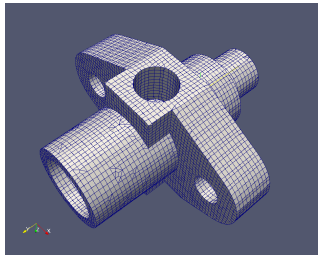


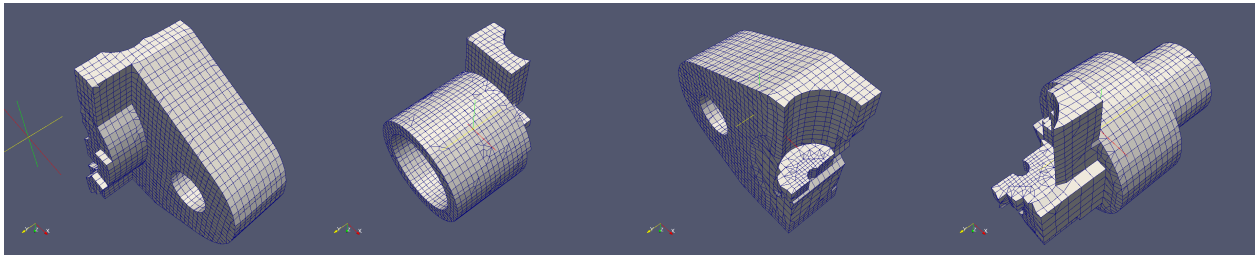

```
Dual center graph
(dual graph)
2 5
1 3
2 4 6
3 5
1 4
3
```

```
Dual vertex graph
(nodal graph)
2 4
1 7 3
2 4 7 8 5
1 3 6
3 6 8
4 5
2 3 8 9
5 3 7 10
7 10
8 9
```

UM: decomposition

- The goal of graph decomposition: *given a graph G(V, E), with vertices V (which can be weighted) and edges E (which can also be weighted), partition the vertices into k disjoint sets such that each set contains the same vertex weight and such that the cut-weight, i.e. the total weight of edges cut by the partition, is minimised.*



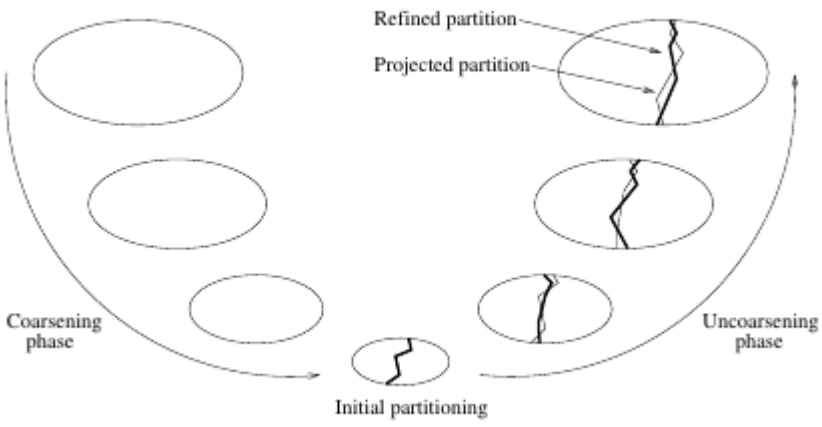


- *More sophisticated goals may be required, e.g.:*
  - *each set hosts only connected cells*
  - *optimization for heterogeneous machines*
  - *subdomain interfaces properties*

Libraries and METIS

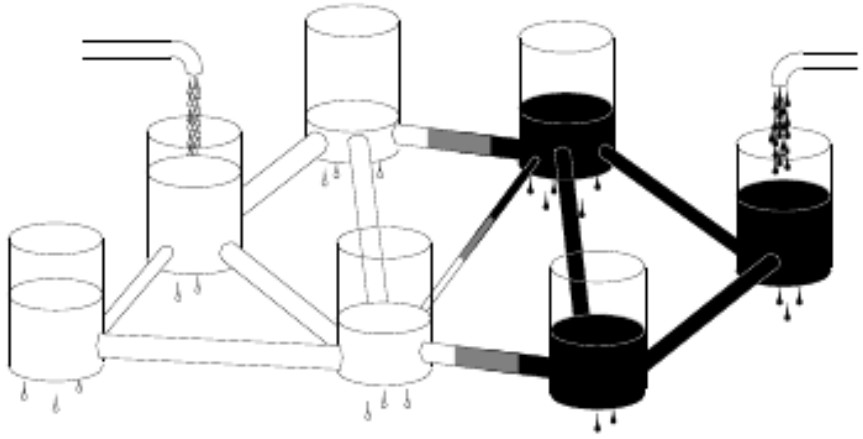
- Decomposition algorithms may be not trivial: use libraries! E.g., METIS, Scotch
  - for huge meshes, mesh decomposition should be parallelized, too
  - again, use libraries! E.g., ParMETIS, Scotch-PT
- Decomposition libraries usually provide stand-alone utilities or APIs
- METIS library allows to convert mesh to dual or nodal graph
  - while other libraries usually lack of this feature

- Best METIS decomposition algorithm is usually the *multilevel k-way partitioning algorithm*
  - *combines global and local optimization approaches*



Scotch

- Implements multilevel banded diffusion scheme

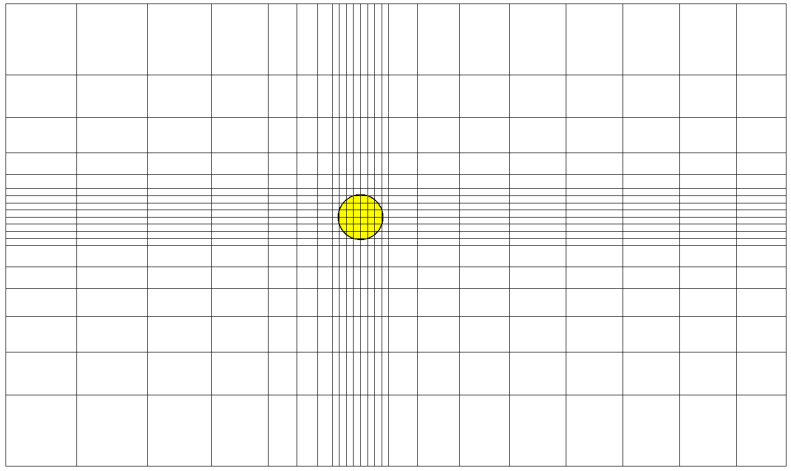


- Compared to METIS, it limits the subdomain area and shape irregularity (useful to fasten iterative convergence)
  - may result in slightly better performances compared to METIS
- The diffusion algorithm is highly scalable (PT-Scotch)

Outline

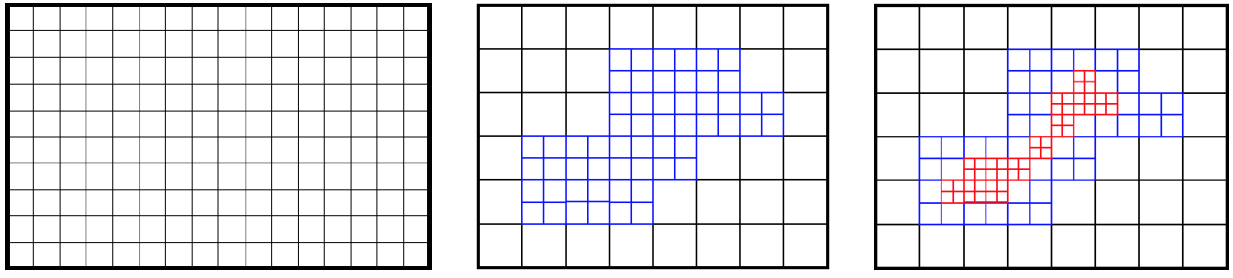
Adaptive Mesh Refinement

- A sophisticated method to handle complex configurations or, in general, useful when a very large range of scales need to be simulated (e.g., CFD, Astrophysics)
- Consider it when non-uniform Cartesian or curvilinear meshes are not enough
  - non-uniform meshes allow decreasing the amount of grid points but the Cartesian (or other) structure limits the achievable reduction

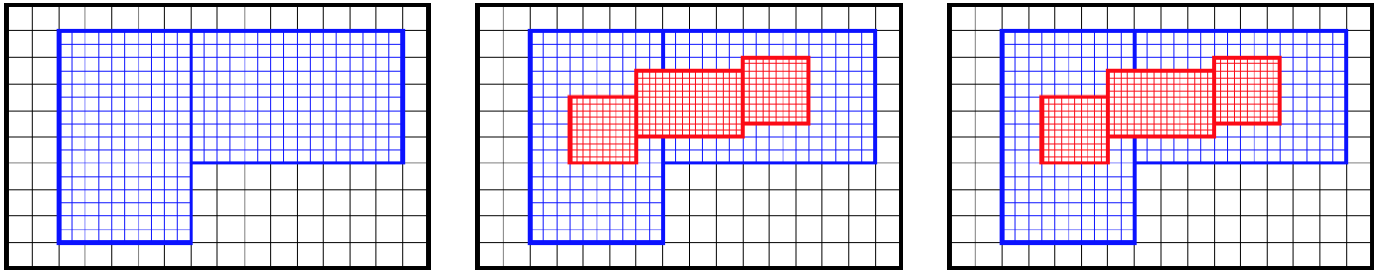


AMR: flavours

- Idea: start with a very coarse mesh and refine it where required: high gradients, close to boundary, . . .
- Different flavours: based on points/patches/blocks
  - point based



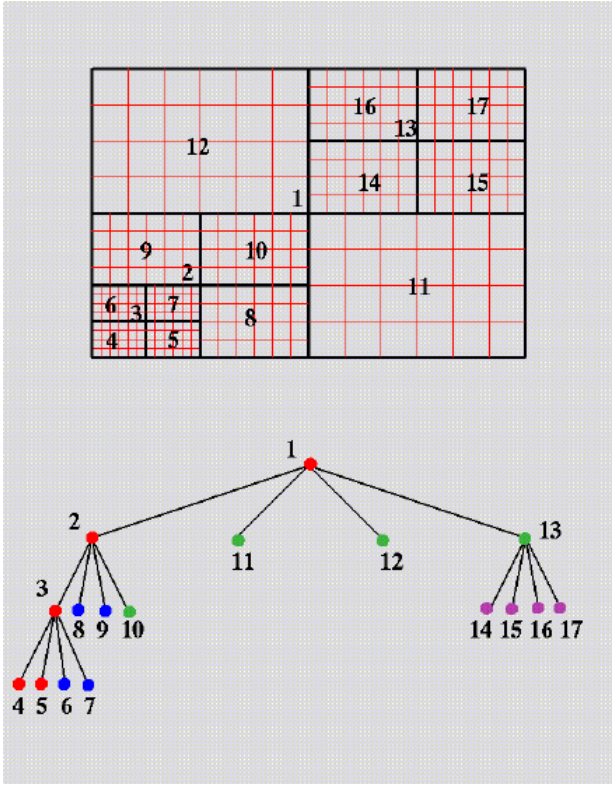
  - patch based



(The figures are Courtesy of Dr. Andrea Mignone, University of Turin)

AMR and trees

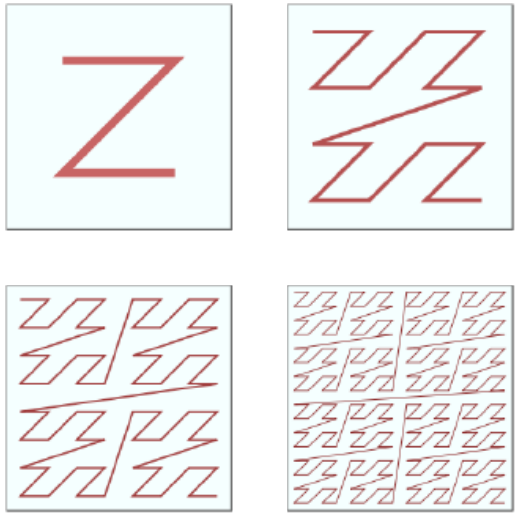
- A common implementation relies on trees (quad-tree or oct-tree)
  - probably not the most efficient: study hash-tables to do better
- Block based tree example

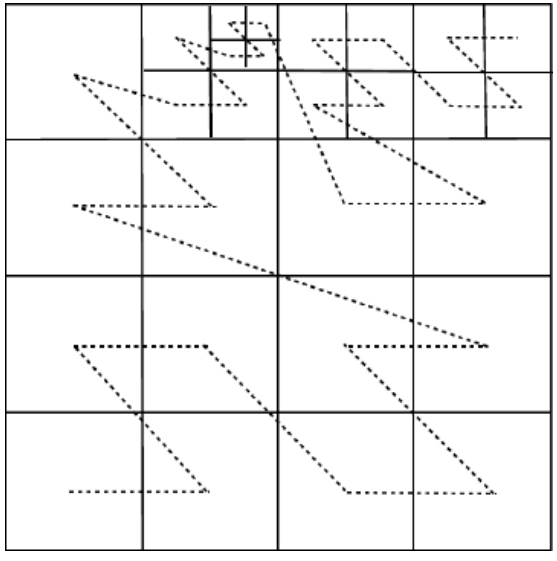


- Hard implementation effort is required
  - managing ghost cells
  - synchronization of patches at the same refinement level
  - interpolation/averaging between different levels
  - evolve only finest grid or all levels
  - block ordering

AMR: block ordering

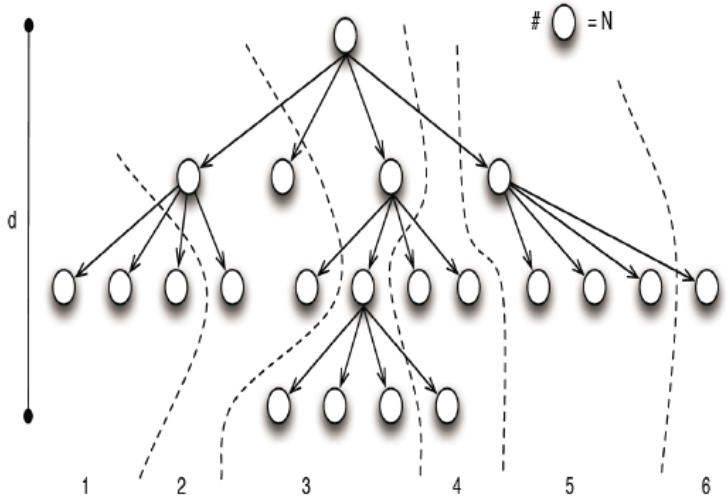
- An additional block ordering algorithm is strongly recommended
  - to optimize the usage of cache memory
  - to optimize ghost cells communications between processes
- Space-filling curves allow to do that
  - Morton or Hilbert algorithms are common choices

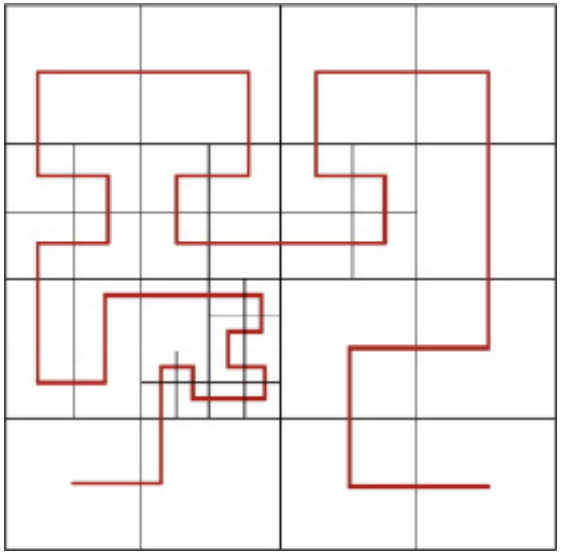




AMR: load balancing

- Using space-filling curves helps when splitting the work-load between processes
  - consecutive points along the curve are physically close
  - the work-load decomposition becomes a one-dimensional decomposition along the curve





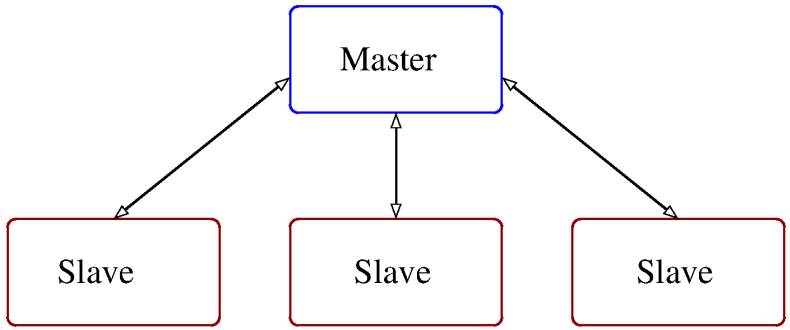
AMR: libraries

- Before re-inventing the wheel check if one of the existing AMR libraries is to your satisfaction
  - PARAMESH - http://www.physics.drexel.edu/ olson/paramesh
  - SAMRAI - https://computation.llnl.gov/casc/SAMRAI/
  - p4est - http://www.p4est.org/
  - Chombo - https://commons.lbl.gov/display/chombo/Chombo

Outline

Static *vs Dynamic Balancing*

- Static load balancing may be not enough
  - e.g., Adaptive Mesh Refinement, global and local amounts of grid points change over time
- When the work-load is unpredictable and synchronizing tasks may become impossible, the master-slave approach may be preferable
  - the master process organizes tasks assigning these to the slave processes
  - the communications occur (only) trough master process



Master-slave example

- The master process:
  - reads from a file a task to be performed
  - waits until a slave process sends a message communicating that it is ready
  - by the way, testing the message (without receiving it) may be useful, **`MPI_Probe`** allows to do it

```
MPI_Probe( int source, int tag, MPI_Comm comm, MPI_Status *status)
```

  - sends the infos about the task to the selected ready slave process

- A slave process:
  - receives infos about the task to be performed
  - performs its task
  - when finished sends back a message to the master

Outline

References

- Algorithms: “Introduction to Algorithms”, T.H. Cormen, Charles E. Lesiserson et Al.
- Finite-Difference: http://www.dtic.mil/dtic/tr/fulltext/u2/a227105.pdf
- Advanced MPI: http://www.training.prace-ri.eu/uploads/tx_pracetmo/advancedMPI.pdf
- Multi-block: http://www.nas.nasa.gov/assets/pdf/techreports/2003/nas-03-007.pdf
- Finite-volumes (OpenFOAM): http://www.linksceem.eu/ls2/images/stories/..... .....Handling_Parallelisation_in_OpenFOAM_-_Cyprus_Advanced_HPC_Workshop_Winter_2012.pdf
- Unstructured grids: http://www.hector.ac.uk/cse/reports/unstructured_partitioning.pdf
- METIS Library: http://glaros.dtc.umn.edu/gkhome/views/metis
- Scotch Library: http://www.labri.fr/perso/pelegrin/scotch/

Rights & Credits