2d_DG_advection Documentation

Release 1.0

Shiqi_He

Dec 12, 2019

CONTENTS:

CHAPTER

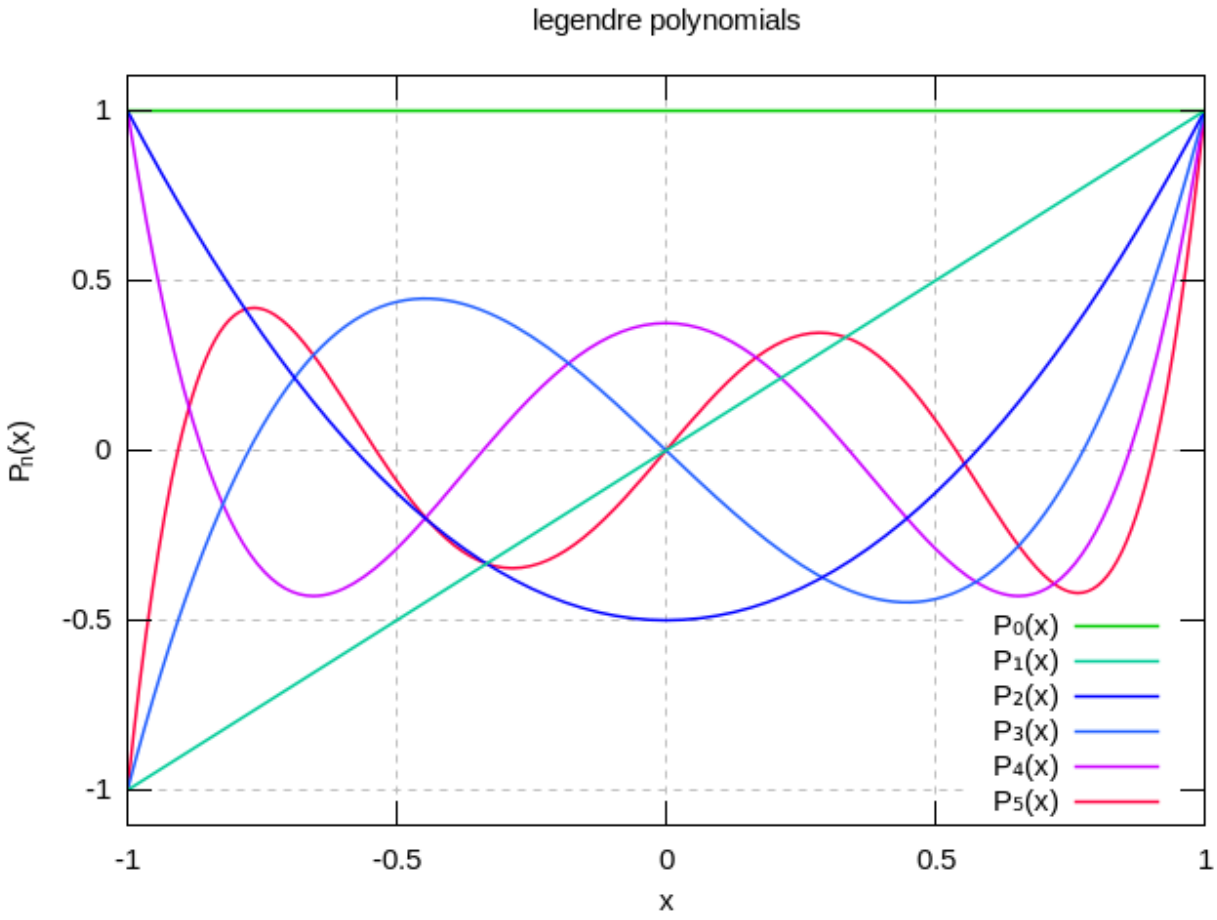
ONE

APPROXIMATING WAVE PROPAGATION

Motivation:

When solving a partial diferencial equation numerically, one has quite a number of difference methods of doing so. Three most widely used numerical methods are finite difference (FDM), finite volume (FVM) and finite element method (FEM). They are different techniques to discerert spatial derivatives. And combine them with an time integration method of an ordinary differential equation, we are able to advance the equation in time.

Finite difference method, though it is simple and intuitive, it has the weakness to handle local one-dimensional polynomial. Moreover, when discontinuous interal layers (e.g., discontinuous material coefficients) are involved, or complicated geometric is needed, the method becomes ill-suited. If we want to combine the geometric flexibility (finite volume) and high-order accuracy (finite element) , Discontinuous Galerkin method is one of your ideal choice.



Discontinuous Galerkin, or DG, overcomes the limitation on achieving high-order accuracy on general grids, compared with FVM. Whereas DG's structure is quite similar to FEM, it's mass matrix is local rather than global, thus, it is less costly to invert that. Additionally, the numerical flux is designed to reflect the underlying dynamics, one has more choices than FEM to ensure the stability for wave dominated problem.

Here, we present a 2D DG solver for a classic wave propagation problem.

1.1 Motivation

Hello world!

1.2 Spectral Approximation

1.2.1 Polynomial Basis Functions

The Legendre Polynomials.

1.2.2 Polynomial Series

1.2.3 Gauss Quadrature

Hello

1.3 Spectral Approximation on a square

1.3.1 Approximation of Wave Propagation

Basic Model

The basic model is the linear wave equation with the form:

$$\frac{\partial^2 p}{\partial t^2} - c^2(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}) = 0$$

The wave equation is the fundamental equation of acoustics. It is based on two improtant approximation, namely, that the flow may be treated as *inviscide* and that *convective derivatives are negligible in comparison to unsteady derivatives*. (we neglect viscous and other diffusion effect(heat), when convection transfer is much faster than diffusion transfer of mass, momentum or energy.)

The variable p may represent acoustic pressure in an otherwise quiescent gas and c could be sound speed.

In order to solve the second order equation, we re-write the equation as a system of three first order equations.

Convert the wave equation to a system of first order equation, let:

$$u_t = -p_x, v_t = -p_y.$$

u and v correspond to the components of the velocity in a fluid flow.

Assuming the order of mixed partial derivatives does not matter, then:

$$\frac{\partial^2 p}{\partial t^2} + c^2((u_x)_t + (v_y)_t) = 0.$$

Combining with initial conditions,

$$p_t + c^2(u_x + v_y) = 0.$$

We now obtain the system of equations by grouping the equation for pressure and two velocity components

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix}_t + \begin{bmatrix} 0 & c^2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix}_x + \begin{bmatrix} 0 & 0 & c^2 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix}_y$$

or

$$\mathbf{q_t} + A\mathbf{q_x} + B\mathbf{q_y} = 0$$

Since A and B are constants, we can bring them inside the derivatives

$$\begin{aligned} \mathbf{q_t} + \mathbf{f_x} + \mathbf{g_y} &= 0 \\ \mathbf{f_x} &= A\mathbf{q_x} \\ \mathbf{g_y} &= B\mathbf{q_y} \end{aligned}$$

This is known as **Conservation law** form since it can be written as

$$\mathbf{q_t} + \bigtriangledown \cdot F = 0$$

where the vector flux $F = \mathbf{f}\widehat{x} + \mathbf{g}\widehat{y}$.

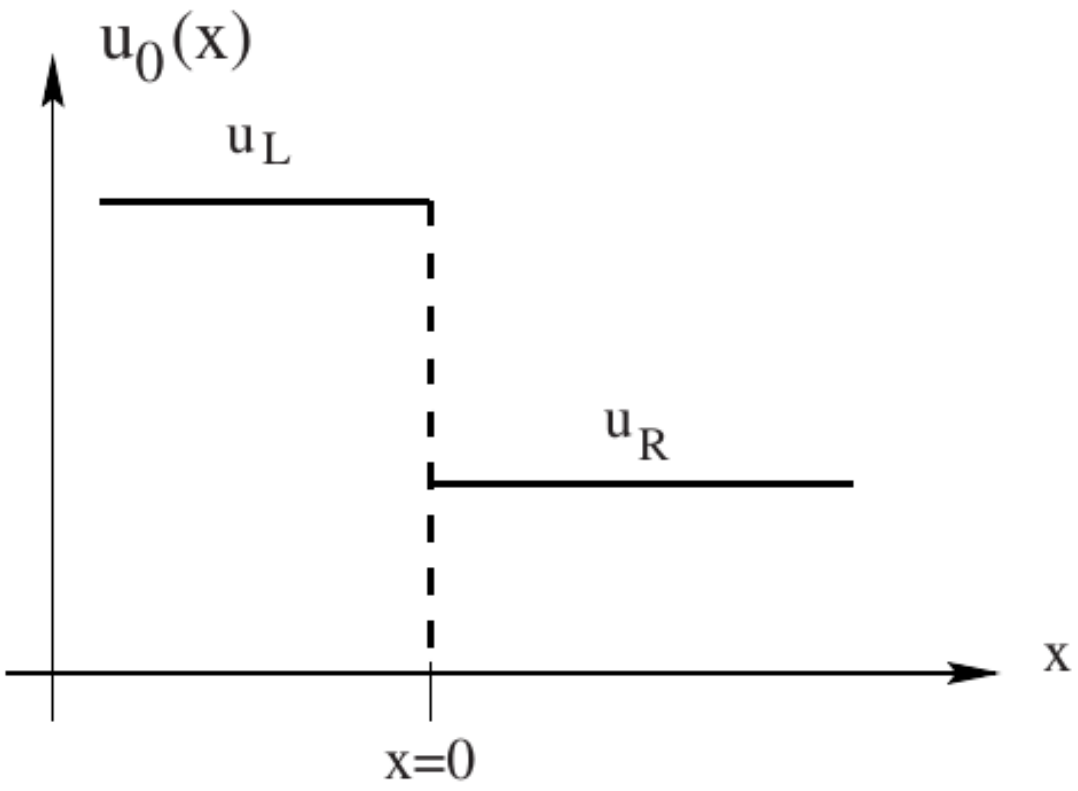
The term conservation law follows from the fact that the differential equation is what we get when we apply the divergence theorem to the integral conservation law.

$$\frac{d}{dt}\int_V \mathbf{q} dV = -\int_S F \cdot \widehat{n} dS$$

Riemann Problem for Conservation Law

Introduction

A Riemann problem, named after Bernhard Riemann, is a specific initial value problem composed of a conservation equation together with piecewise constant initial data which has a single discontinuity in the domain of interest. The Riemann problem is very useful for the understanding of equations like Euler conservation equations because all properties, such as shocks and rarefaction waves, appear as characteristics in the solution. It also gives an exact solution to some complex nonlinear equations, such as the Euler equations.



Riemann Solver

Here we build a Riemann problem for the hyperbolic, constant coefficient system with proper initial condition.

$$\mathbf{q_t} + A\mathbf{q_x} + B\mathbf{q_y} = 0$$

The coefficient matrices A and B have m real eigenvalues λ_i and m linearly independent eigenvectors $\mathbf{K}^{(i)}$, where m is the equation number *Reference*.

The Nodal Discontinuous Galerkin Approximation

We will implement the discontinuous Galerkin spectral element approximation of two-dimensional conservation law on a square domain.

$$\mathbf{q_t} + \mathbf{f_x} + \mathbf{g_y} = 0, x \in (L, R), y \in (D, U) \tag{1.1}$$

The spectral element approximation starts with a weak form of (1.1). We multiply (1.1) by a test function, integrate and subdivide into elements

$$\sum_{k=1}^{K} \left[\int_{x_{k-1}}^{x_k} (\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y)\phi dx \right] = 0 \tag{1.2}$$

We map (1.2) onto reference space by **affine map** (1.3)

$$\begin{aligned} x &= x_{k-1} + \frac{\xi+1}{2}\Delta x_k, \Delta x_k = x_k - x_{k+1} \\ y &= y_{k-1} + \frac{\eta+1}{2}\Delta y_k, \Delta y_k = y_k - y_{k+1} \\ dx &= \frac{\Delta x_k}{2}d\xi, \frac{\partial}{\partial x} = \frac{2}{\Delta x_k}\frac{\partial}{\xi} \end{aligned} \tag{1.3}$$

The solution and fluxes are approximated by polynomials of degree N and represent the polynomials in nodal, Lagrange form

$$\begin{aligned} \mathbf{q} \approx \mathbf{Q} = \sum_{n=0}^{N}\sum_{m=0}^{M}\mathbf{Q}_{n,m}l_n(x)l_m(y) \\ \mathbf{F}_{n,m}\widehat{x} + \mathbf{G}_{n,m}\widehat{y} = B\mathbf{Q}_{n,m}\widehat{x} + C\mathbf{Q}_{n,m}\widehat{y} \end{aligned} \tag{1.4}$$

where $\mathbf{F}_{n,m}\widehat{x} + \mathbf{G}_{n,m}\widehat{y} = B\mathbf{Q}_{n,m}\widehat{x} + C\mathbf{Q}_{n,m}\widehat{y}$. We subsitute the approximations into the weak form of the PDE, and let $(\mathbf{Q}_t, \phi_{ij}) + (\bigtriangledown \cdot \mathbf{F}, \phi_{ij}) = 0$.

If we apply Green's identity to the second intergal

$$(\bigtriangledown \cdot \mathbf{F}, \phi_{ij}) = \int_l^r \phi_{ij} \bigtriangledown \cdot \mathbf{F} dxdy = \frac{\Delta x}{2}\int_{-1}^{1}\phi_{ij}\mathbf{f}_\xi d\xi + \frac{\Delta y}{2}\int_{-1}^{1}\phi_{ij}\mathbf{g}_\eta d\eta$$

The Nurmerical flux

Time Integration

Change of Interval

Benchmark Solution: Plane wave Propagation

We represent a plane Gaussian wave through the grid.

The plane wave is defined as:

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{k_x}{c} \\ \frac{k_y}{c} \end{bmatrix} e^{-\frac{(k_x(x-x_0)+k_y(y-y_0)-ct)^2}{d^2}}$$

Where $\mathbf{k}$ is the wavevector and it is normalized to satisfiey $k_x^2 + k_y^2 = 1$. The wavevector is choosen as $\mathbf{k} = (\sqrt{2}/2, \sqrt{2}/2)$ This is a wave with Gaussian shape where we compute the parameter d from the full width at half maximum, $\omega = 0.2$, by math:*d = omega/2sqrt{ln2}*. The other parameters are $c = 1$ and $x_0 = y_0 = -0.8$.

Performance Evaluation

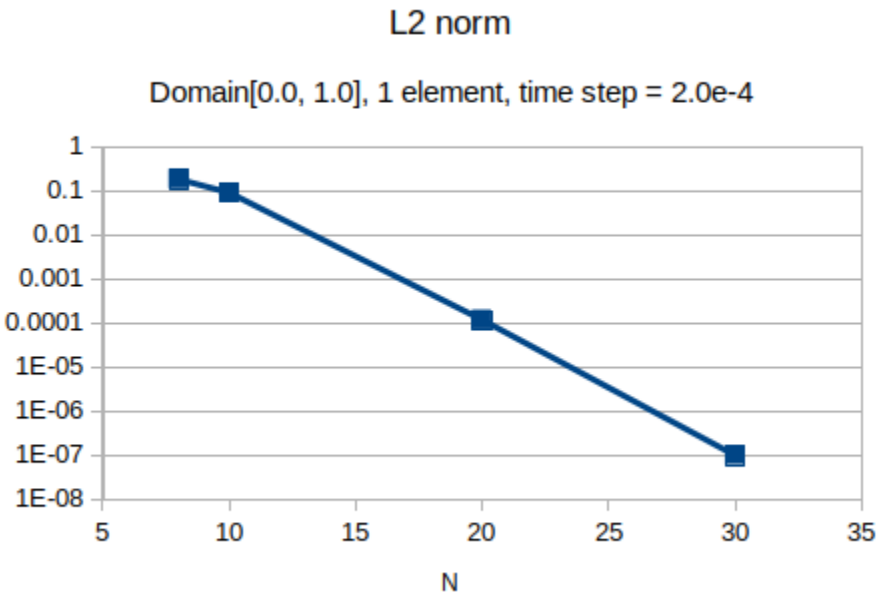
Exact boundary solutions are imposed on the 4 side of the computation domain. The initial condition is setting *t=0.0* of the exact solution.

1 element

Domain: $x \in [0.0, 1.0], y \in [0.0, 1.0]$.

Time step: $\Delta t = 2.0 \times 10^{-4}$

Fig(1), shows the error performances.

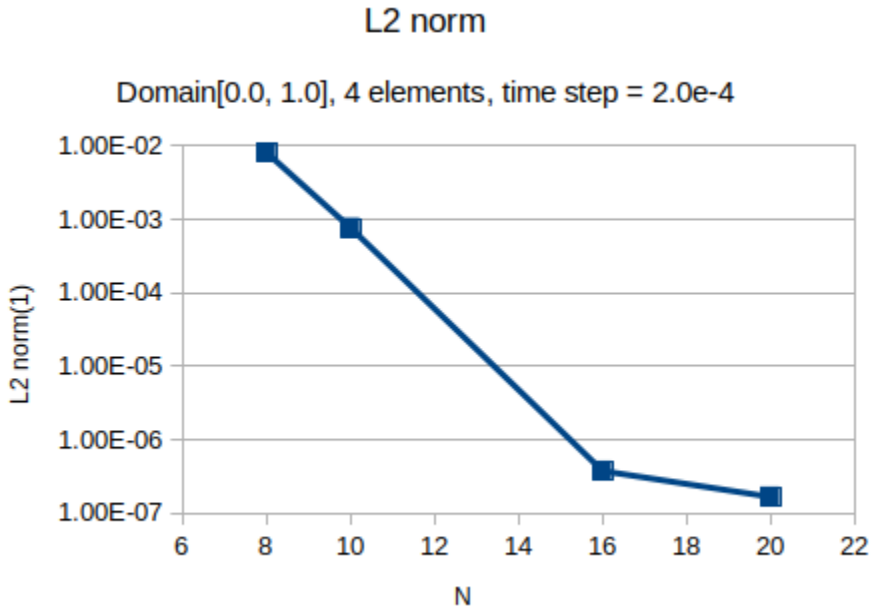


4 element2

Domain: $x \in [0.0, 1.0], y \in [0.0, 1.0]$.

Time step: $\Delta t = 2.0 \times 10^{-4}$

Fig(2), shows the error performances.

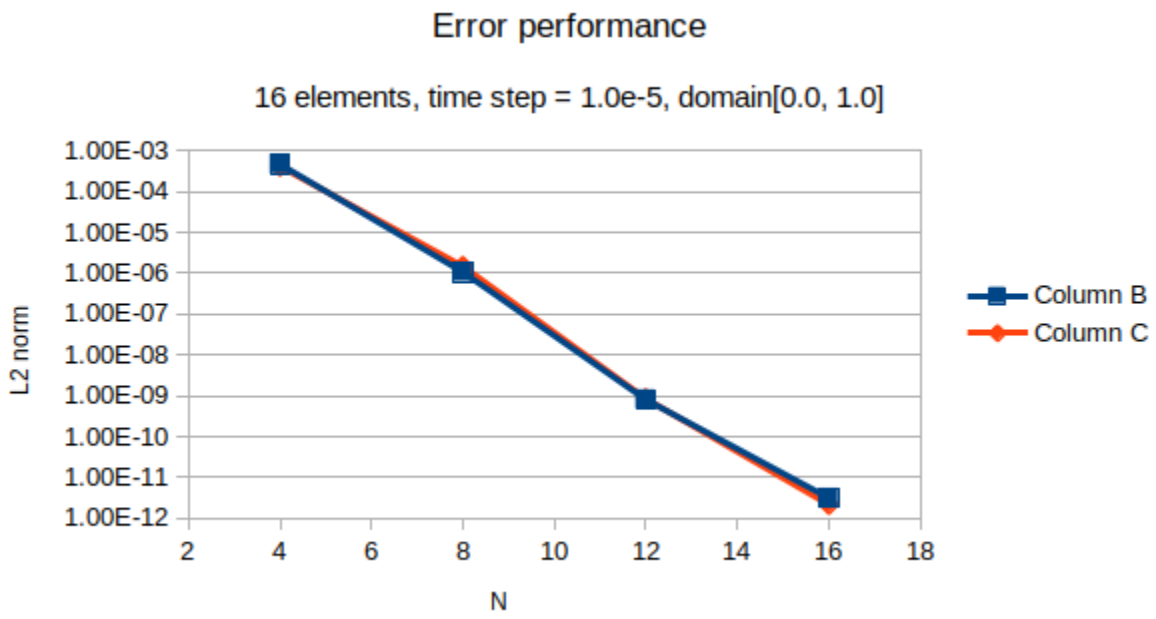


16 elements

Domain: $x \in [0.0, 1.0], y \in [0.0, 1.0]$.

Time step: $\Delta t = 1.0 \times 10^{-5}$

Fig(3), shows the error performances.

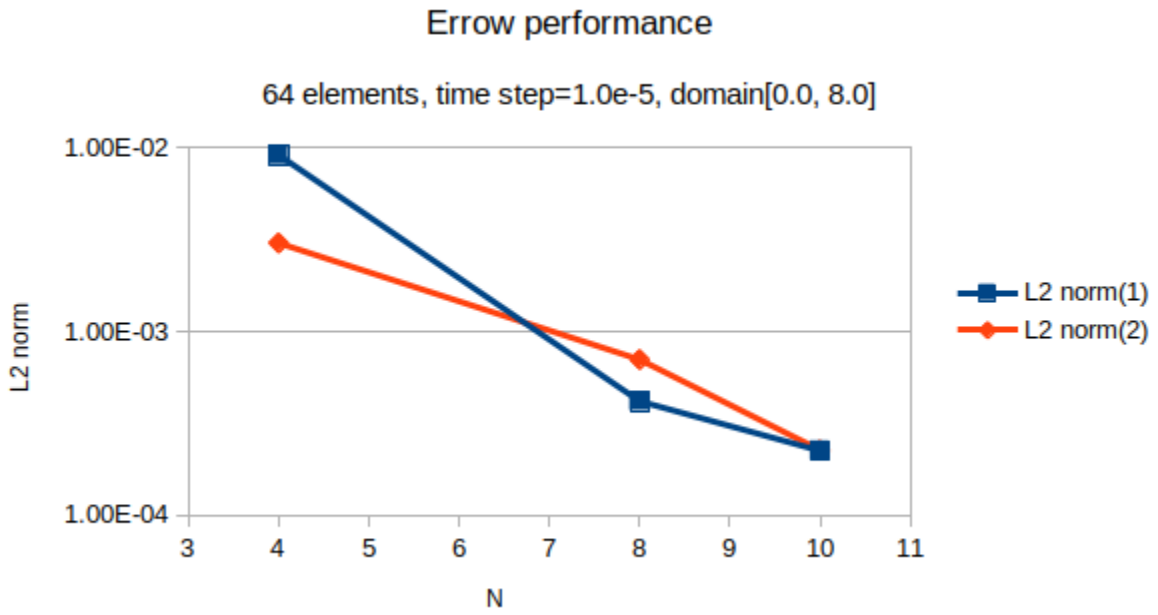


64 elements

Domain: $x \in [0.0, 8.0], y \in [0.0, 8.0]$.

Time step: $\Delta t = 1.0 \times 10^{-5}$

Fig(3), shows the error performances.



1.4 Numerical Flux schemes

1.4.1 Central Flux

$a + b = 1$

$a + b = 1(1.5)$

1.4.2 Lax-Friedrichs Flux

1.4.3 Upwind Flux

1.5 AMR Strategies

1.5.1 Introduction

Why AMR?

In order to effectively utilize the computational resources while remaining the flexibility in solving complex geometries and the prescribed accuracy, **Adaptive Mesh Refinement (AMR)** is invoked to focus the computational effort and memory usage to where it is needed.

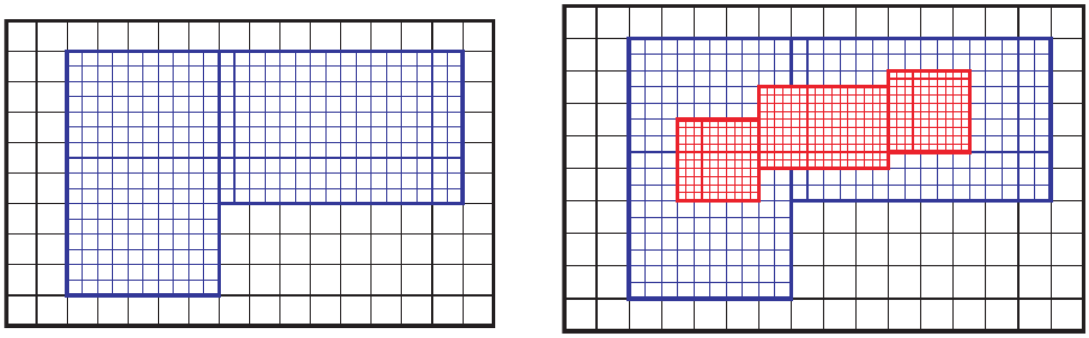
Three main algorithms

Three main algorithms have emerged overtime, which we can call them: **unstructured (U), block-structured (s)**, and hierarchical or **tree-based (T)** AMR.

UAMR

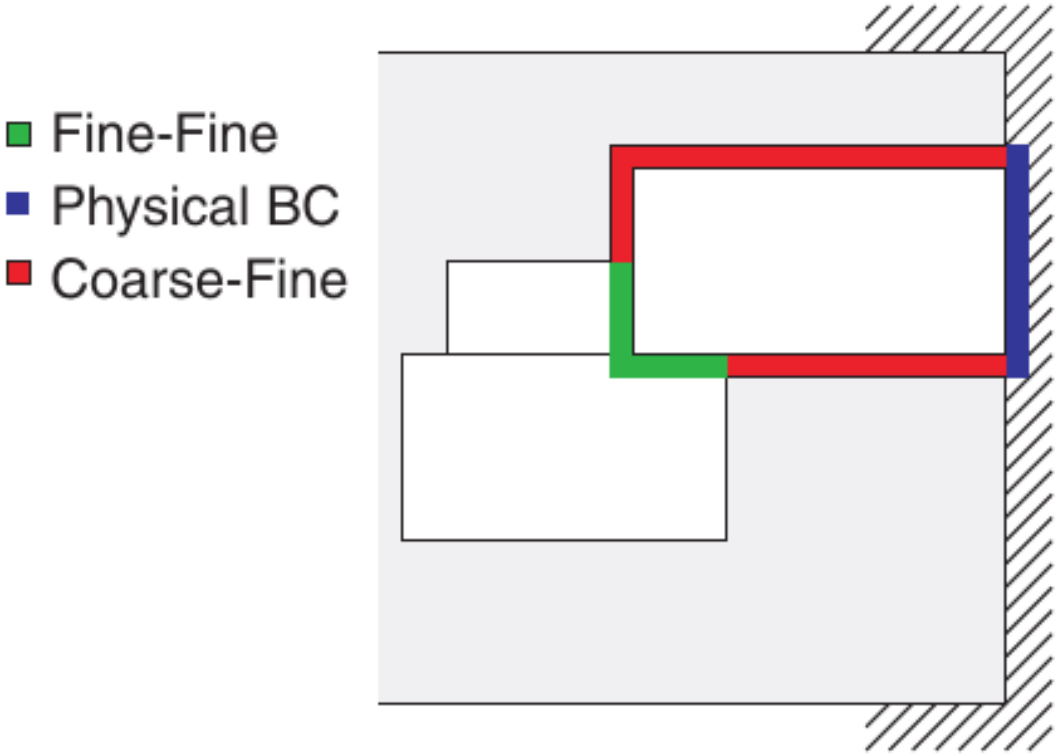
Unstructured mesh. Traditionally use graph-based partitioning algorithm, now are supplementing by fast algorithms based on coordinate partitioning and SFCs.

SAMR



A sequence of nested structured grids at different hierachies or levels are overlapped with or patched onto each other.

A tree-like data structure is used to facilitate the communication (transfer information) between the regular Cartesian grids at the various hierachies. Each node in this tree-like data structure represents an entire grid rather than simply a cell.



Pros:

- Each node in the tree structure represents an entire grid enables the solver to solve the structured grids efficiently.

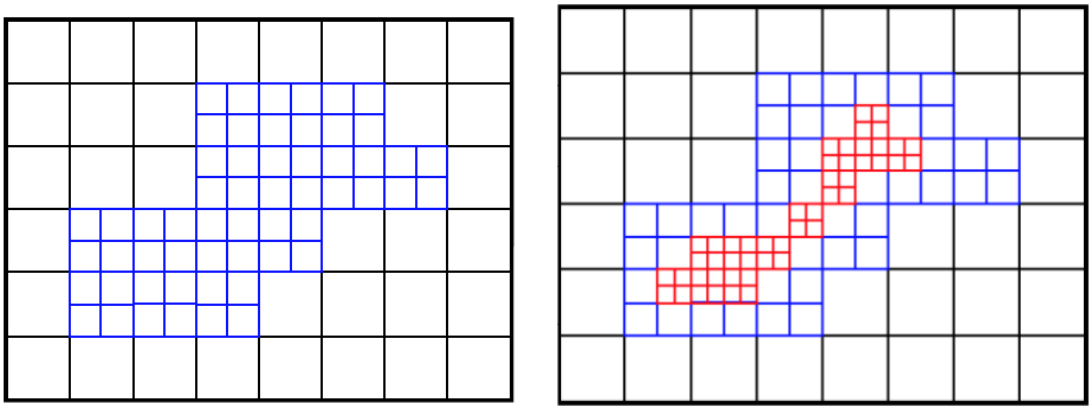
Cons:

- Communication patterns between levels can be complex.
- Algorithm complexity can be substantial.
- Due to the clustering methods used to define the sub-grids, portions of the cumputational dmain covered by a highly refined mesh when it is not needed, resulting a wasted computational effort.

Library

- Chombo
- PARAMESH
- SAMRAI

TAMR



A qurad-tree/oct-tree data structure is used in 2D/3D to represent the grid hierarchies. Each node stands for a individual cell.

Pros:

- Mesh can be locally refined (increase storage savings)
- Better control of the grid resolution (comparing with SAMR)

Cons:

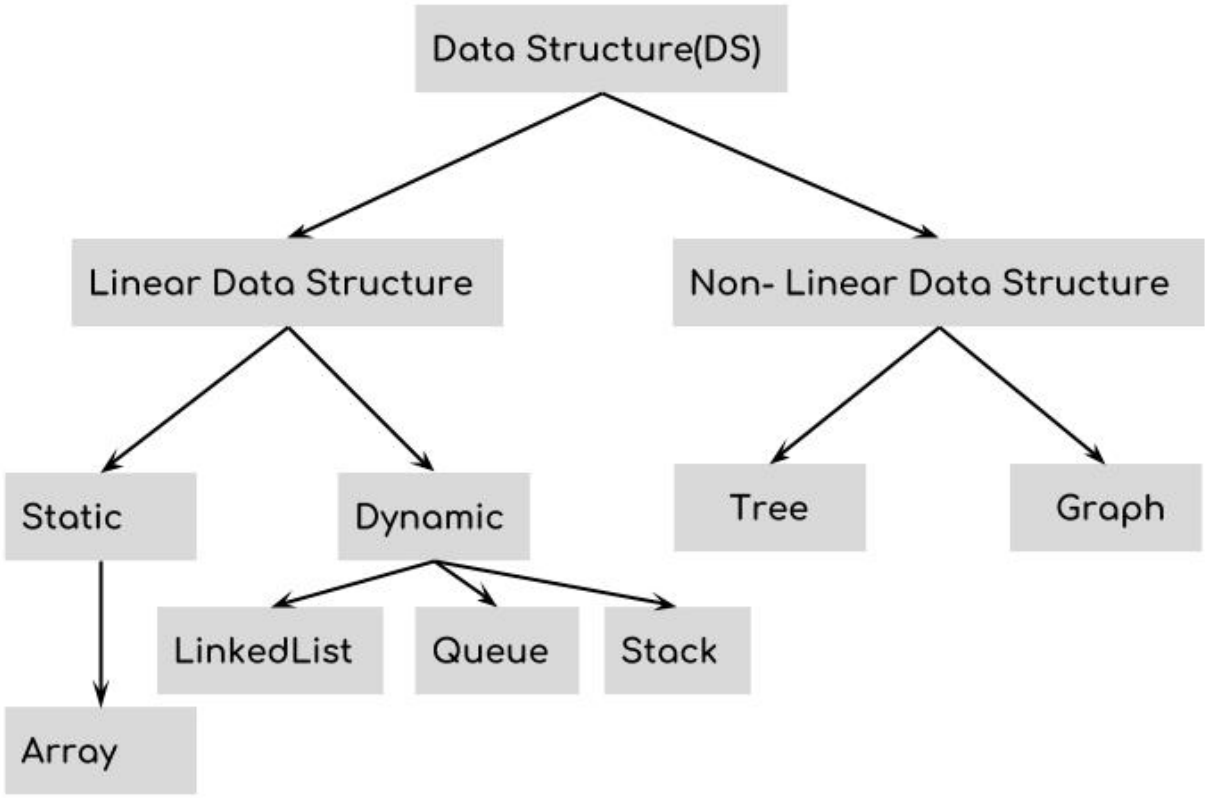
- In conventional quard-tree/oct-tree discretization, the connectivity information between individual cell and its neighbours needs to be stored explicitly. (oct-tree each cell 19 words of computer memory)
- large memory overhead to maintain tree-data structures.
- **Difficult to parallelize.**
  - data moving: distruct and rebuild the linker.
  - neighbour finding: need to traverse the tree to locate the closet ancestor (what if ancestor is on another processor?).

Library

- p4est
- Zoltan

1.5.2 Data structure: Quardtree/Octree

Data structure Classifications



Quardtree definition

A quadtree is a tree data structure in which each internal node has exactly four children. Quadtrees are the two-dimensional analog of octrees and are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions.

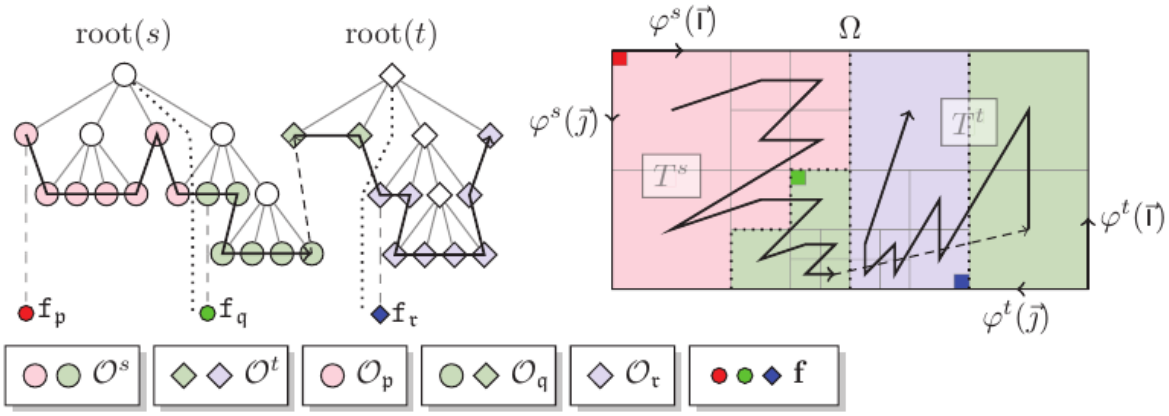
1.5.3 Tree-based AMR algorithm

Objectives

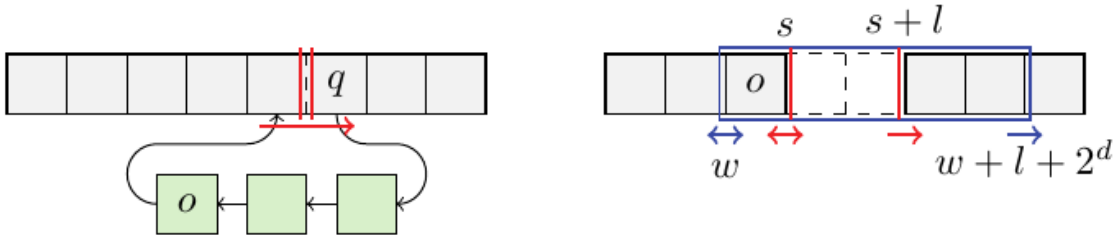
- Reduce the memory overhead required to maintain the information embodies in the tree structure.
- Rapid and easy access to the information stored in the tree.

p4est

Linear octree



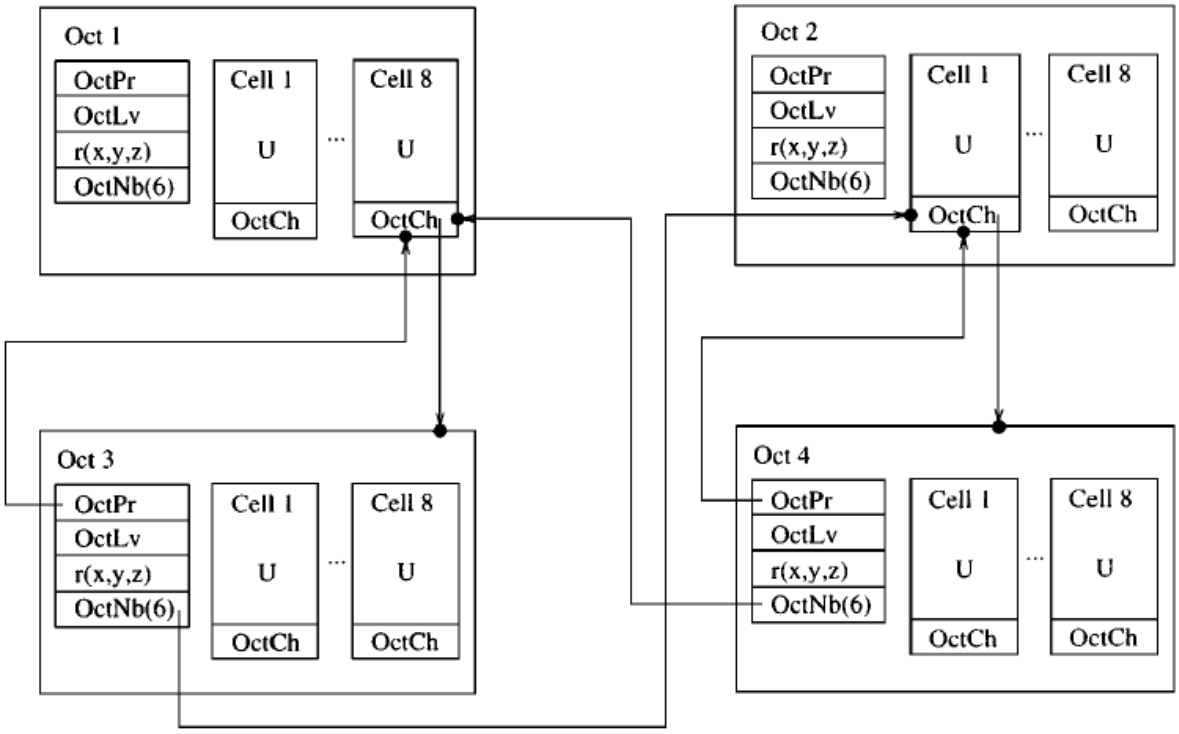
Only store the leaves of the octree (“linear” octree).



Full Threaded Tree (FTT)

Memory requirement: $2\frac{3}{8}$ words per cell (conventional 19 words per cell).

The actual number of traversed levels required to find a neighbour never exceeds one.

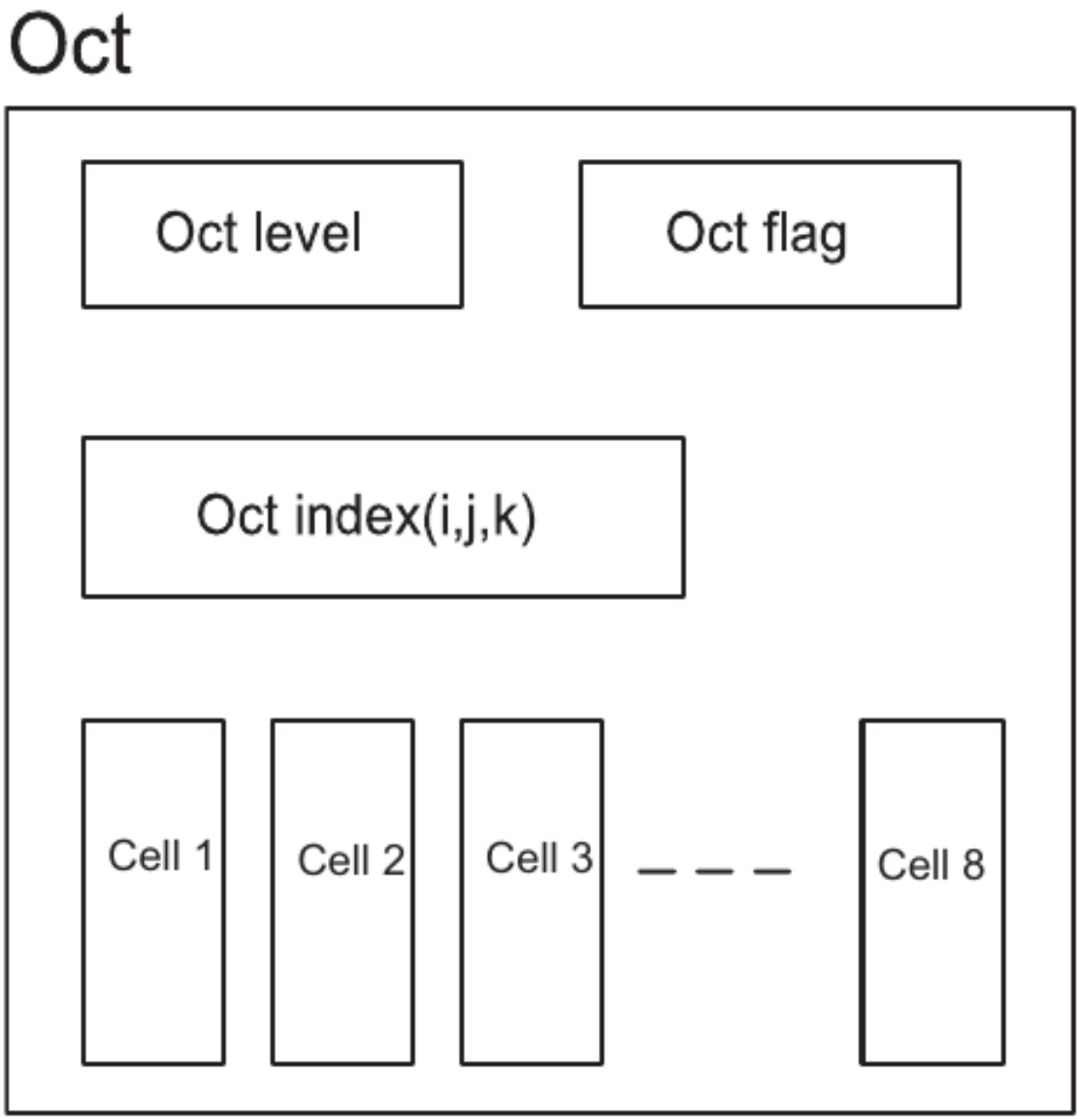


Cell-Based Structured Adaptive Mesh Refinement

Optimized FTT.

Cartesian-like indices are used to identify each cell. With these stored indices, the information on the parent, children and neighbours of a given cell can be accessed simply and efficiently.

Memory requirement: $\frac{5}{8}$ words per cell.



Octant coordinate calculation

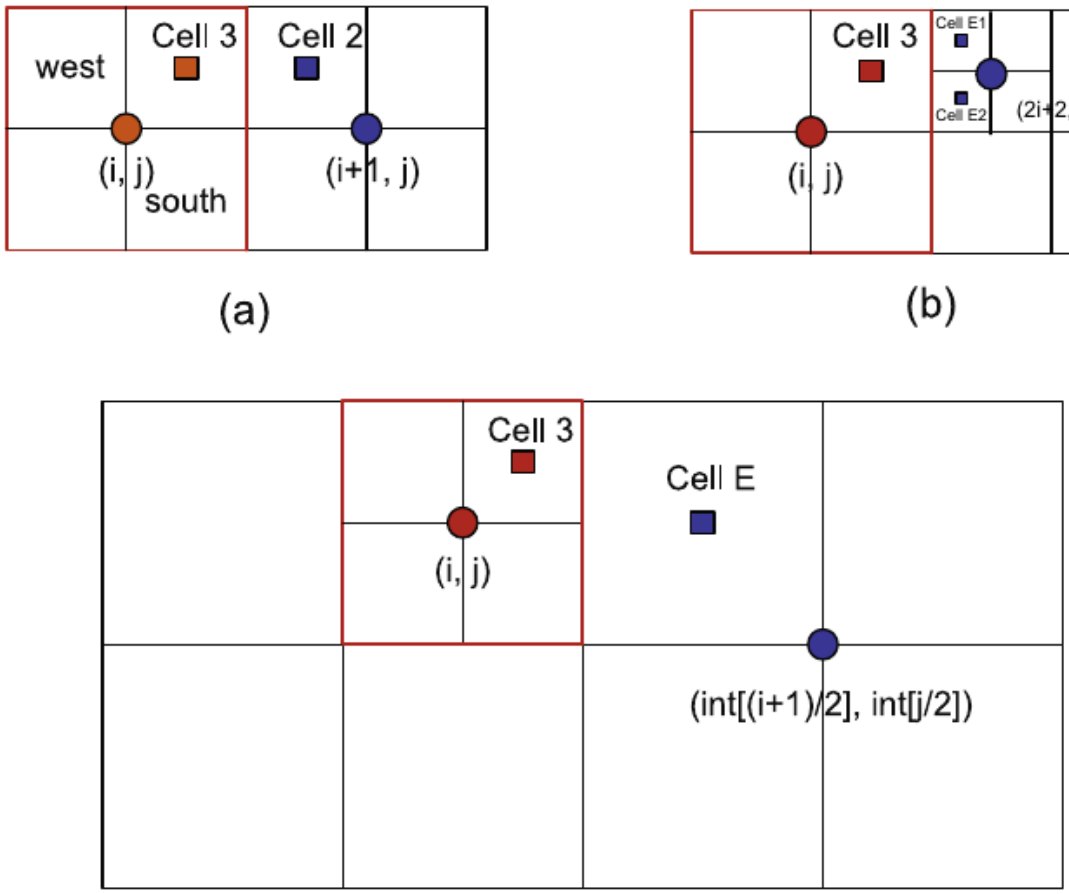
The indices of the four children octs (i_s, j_s)

$$(i_s, j_s) = \{(2i + m, 2j + n)|m = 0, 1; n = 0, 1\}$$

The parent of a oct (i_p, j_p)

$$(i_p, j_p) = \left(int[\frac{i}{2}], int[\frac{j}{2}]\right)$$

Neighbour finding



Cell3 find east neighbour:

(1). (i+1, j) – hash table – cell exsit (Y/N)?

(2). If Yes.

- Neighbour is the Northwest (NW) cell of cell(i+1, j) – if this cell is a leaf (Y/N)?
  - Yes – over
  - No – two neighbours (NW, SW)

(3). If No.

- Neighbour cell has a larger size. cell number is $\left(int[\frac{i+1}{2}], int[\frac{j}{2}]\right)$

At most, two search are sufficient to find a neighbour of a give cell. Half of the neighbours can be reached without consulting the hash table. Statistically, the average number of searches required to find a neighbour of a given cell is one.

1.6 Dynamic Load-balancing

1.6.1 Motivation

Load balance is one of the major challenges for the efficient supercomputer, especially for applications that exhibit workload variations. Load imbalance is an considerable impedance on the path towards higher degree of parallelism.

In particular, when load conditions changes dynamically, efficient mesh partitioning becomes an indispensible part of scalable design.

1.6.2 Goals

Fluid dynamic application in the field of industrial engineering require high degrees of parallelism to achieve an acceptable time to solution for a large problem size. Typical mesh-based approaches therefore rely on suitable partitioning strategies to distribute the computational load across the set of processes.

Therefore, an efficient load-balancing approach aims to achieve two goals:

- **The work load should be distribute evenly**
  - avoid waiting times of processing units
- **At the same time the interfacing boundaries between partitions should be as small as possible.**
  - minimize the time spend in communication

The optimization problem is **NP-hard**.

1.6.3 Two popular approaches

Graph-based Algorithm

A popular choice for graph-based partition is ParMetis.

ParMetis performing very well for mesh partition for a long time. However, since ParMetis require global knowledge of the mesh, with an increasing number of processes, graph-based partitioning algorithms seem to reach their scalability limits. The memory consumption grows linearly with the graph size, raising the need for alternatives which could avoid this problem. Such methods are based on space-filling curves (SFCs).

Space-filling curves (SFCs) based algorithm

SFCs reduce the partitioning problem from n dimension to one dimension. The remaining tast, the so-called 1D partitioning problem or *chains-on-chains* partitioning problem, is to decompose a 1D workload array into consecutive, balanced partitions.

Advantages

- **Good Locality**
  - SFCs map the 1D unit interval onto a higher dimensional space such that neighboring points on the unit interval are also neighboring points in the target space.
- **Acceptable communication overhead**

  - SFCs ignores the edges of full graph imformation. It relies on the spatial properties of the curve to ensure a reasonable partition shape. Tirthapura et al. demonstrated that the upper limit of expected remote accesses in SFC partiotned domains are acceptable [TSA06].

- **Low memory using**
  - Taking the good locality of SFCs, the global imformation (full graph information) needed by Graph-based algorithm can be abandoned. Thus, SFCs opens a path towards low-memory partitioning strategies.

1.6.4 Implementing SFC

The numerical approximation of wave equation is a hp-adaptive approach. That is, elements can split or merge (h-adaptive) according to therequired resolution. Also, they can raise or decrease the polynomial degree (p-adaptive) to adjust the convergence rate.

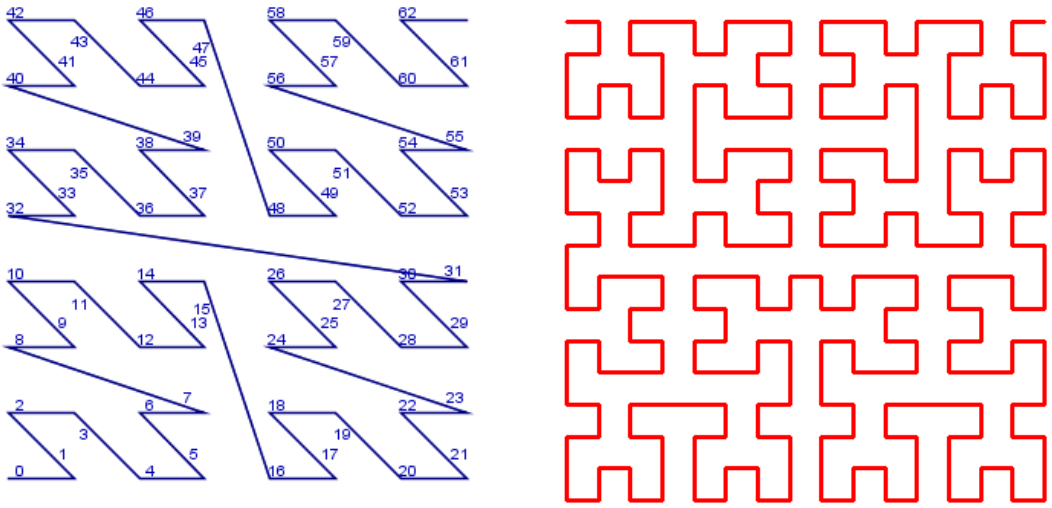
Due to the hp-adaptivity, different element can have differernt individual computation times, load imbalance is introuduce to this application. Re-meshing and domain partitioning are not avoidable.

With the help of a SFC, the 2D structured mesh partitioning problem can be reduced to a 1D chains-on-chains partitioning (CCP) problem. Knowing the index od an element, its neighbours indices can be computed locally.

Hilbert Curve

There are many SFCs, for example Morton Curve (z-curve) and Hilbert Curve.

We choose Hilbert Curve as our SFC. Although Hilbert ordering is less efficient (with flip and rotation) than Morton Curve, Hilbert Curve brings out a better locality (no sudden "jump").



(Left Morton and right Hilbert)

Static Grid Neighbour-finding algorithm

In Computation Fluid Dynamics, most of the cases, elements needs to exchange information (e.g. fluxes, velocity, pressure) with their neighbour. Thus, an effective way to locate your neighbours would cut down the computation time. When the neighbour is not stored locally, communication between processors is inevitable.

For instance, we are on element 31. The domain is partitioned into4 parts and each part is assigned to one processor. The integer coordingate of element 31 is (3, 4).

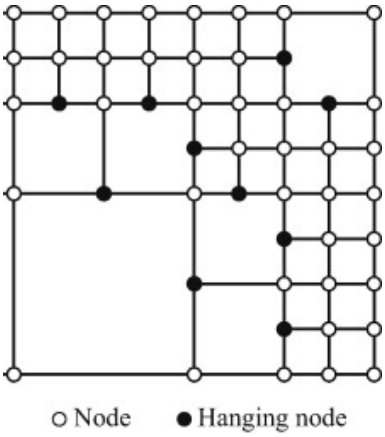
Therefore, its neighbours coordinates can be computed easily. Say we want to find its North and East neighbour, their coordinates are (3, 5) and (4, 4), respectively.

North neighbour: We can use our *Hilbert-numbering function* to map between coordinate and element index. Then (3, 5) corresponding to element 28. We successfully locate the Neighbour.

East neighbour: By using the same methond, we are able to compute the east neighbour index: 32. However, this element is not stored locally. Locate the processor who stores the target element is done by **broadcasting** the element range stored in each processor after the partitioning. And **one-sided communication** is invoked to warrent effective MPI message-changing.

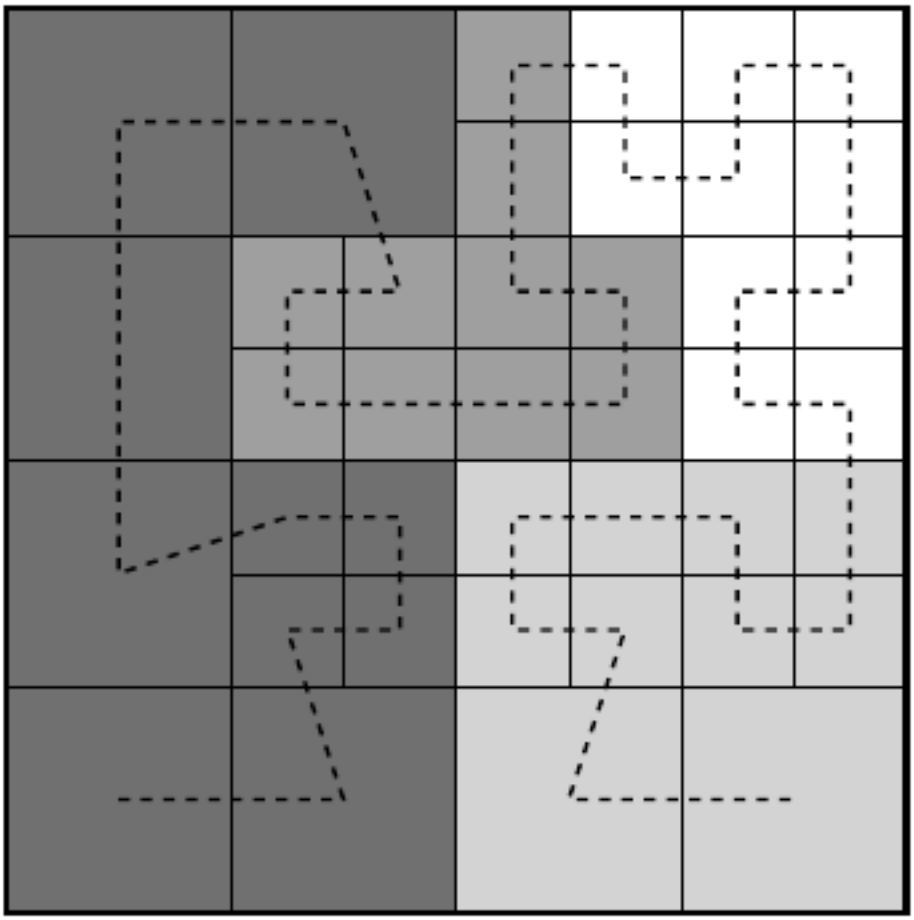
Dynamic grid Neighbour-finding algorithm

When h-adaptivity is introduced to the code, element splits or merge according to the error indicator. Once an element split, it generates four identical "children" quadrants. The **Octree partitioning** is motivated by octree-based mesh generation.



Neighbour-finding is achieved by using a global index (k, l, j, s) to identify element.

- k: Root element number.
- l: h-refinement level (split number).
- j: child relative position inside a parent octant.
- s: element state, can be used to determined Hilbert Curve orientation.

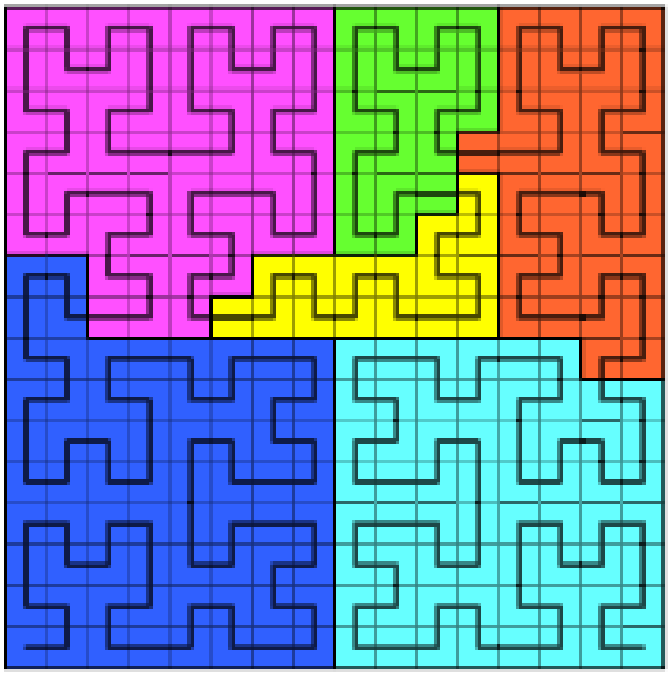


1.6.5 Partitioning stratigy

We consider a 2D mesh being represented by a one dimensional array using Hilbert Curve.

Implementation

We followed the *CCP* strategy described in [HKR+12]. The array has the length N which corresponding to the number of mesh cells. Weights are give as ω_i, where i corresponding to teh global index for each element. The weights represents the computation effort of each element. In fact, the load on each element due to fluid computation is $O(N^4)$[ZBMZ+18].



The task of the partition step is to decide which element to move to which processor. Here, we use p to denote the total number of processors, and every processor can be identified by a unique number called $rank$. ($0 \leqslant rank \leqslant p$)

We use an exclusive prefix sum to determine the partition.

$$prefix(I) = \sum_{i=0}^{N-1} \omega_i \tag{1.6}$$

For $0 < I \leqslant N$ and $prefix(0) = 0$. Local prefix sus are calculated, and the global offsets are adjusted afterwards using *MPI_EXSCAN()* collective with *MPI_SUM* as reduction operation. Then each prossessor has the global prefix sum for each of its local elements.

The ideal work load per partition is given by

$$\omega_{opt} = \frac{\omega_{globalsum}}{p} \tag{1.7}$$

Where $\omega_{globalsum}$ is the global sum of all weights. Since the overall sum already computed through the prefix sum, we can use the last processor as a root to broadcast (*MPI_BCAST*) the ω_{opt}. Then the splitting positions between balanced partitions can be computed locally. There is no need further information changing to decide which element to move to which processor. The complete message changing for the partitioning only relies on two collective operation in *MPI*. Both collectives can be implemented efficiently using asymptotic running time ad memory complexity of $O(logp)$.

Assuming homogeneous processors, ideal splitters are multiples of ω_{opt}, i.e., $r \cdot \omega_{opt}$ for all integer r with $1 \leqslant r < p$. The closest splitting positions between the actual elements to the ideal splitters can be found by comparing with the global prefix sum of each element.

The efficiency E of the distribution work is bounded by the slowest process, and thus cannot better than:

$$E = \frac{\omega_{opt}}{max_{r=0}^{p-1}(\omega_{sum}(r))}$$

Exchange of Element

After the splitting positions are decided, elements needs to be relocated. The relocation, or exchange of elements is done via communication between processors. The challenge part is, though, the sender knows which element to send to which processor, the receiver does not know what they will receive.

Some application use a regular all-to-all collective operation to imform all processors about the their communication partners before doing teh actual exchange of the elements with an irregular all-to-all collective operation (e.g. *MPI_Alltoallv*).

Alternatively, elements can be propagated only between neighbour processors in an iterative fasion. This method can be benigh when the re-partitioning modifies an existing distribution of element only slightly. Unfortunately, worse cases can lead to $O(p)$ forwarded messages.

In our implementation, **One-sided Communication in MPI** is invoked. In one-sided MPI operations, also known as **RDMA** or **RMA** (Remote Memory Access) operation. In RMA, the irregular communication patten can be handle easily without an extra step to determine how many sends-receives to issue. This makes dynamic communication easier to code in RMA, with the help of *MPI_Put* and *MPI_Get*.

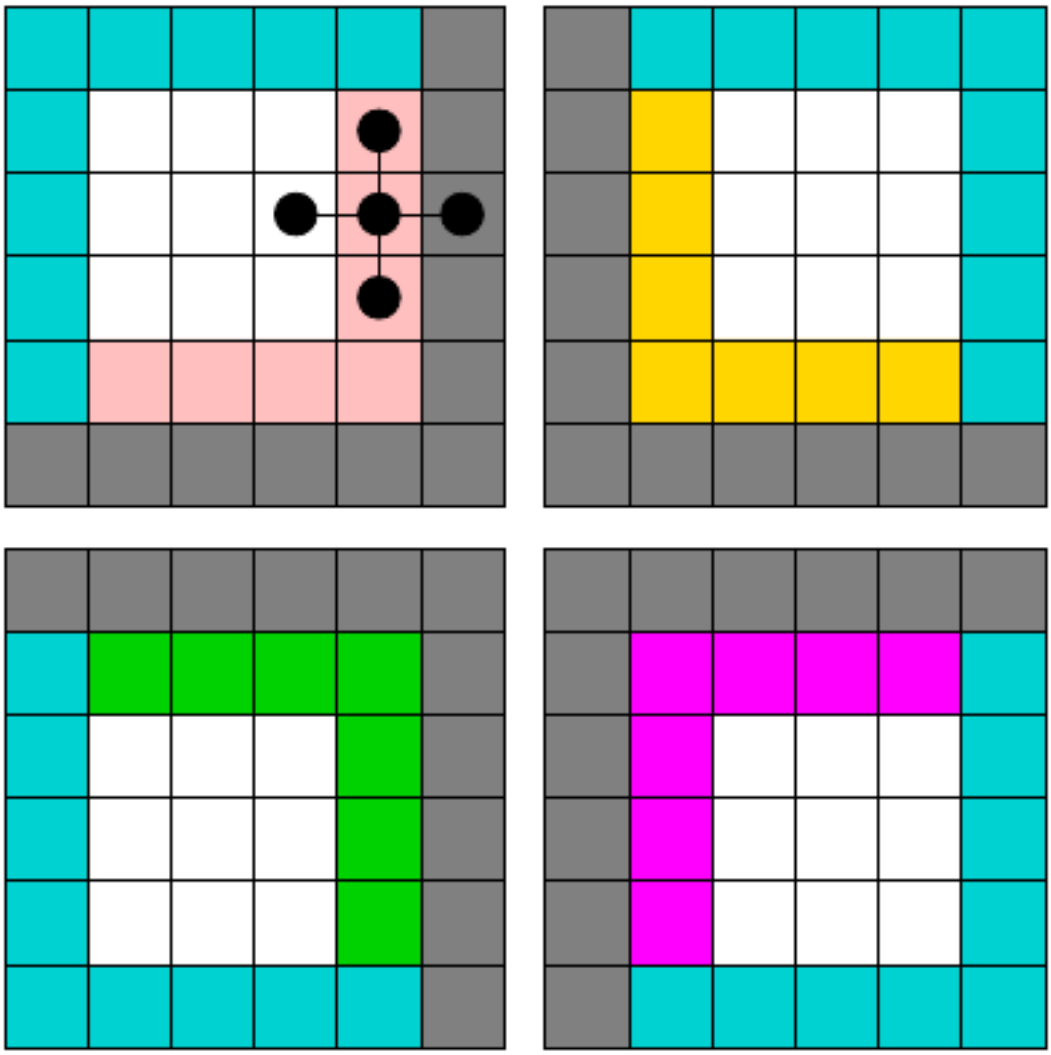
1.6.6 References

1.7 MPI Interface

1.7.1 One-sided Communication in MPI

Motivation

- The receiver does not know how much data to expect (non-conforming).
- Avoid send/recv delay.



Basic Idea

The basic idea of one-sided communication models is to decouple data movement with process synchronization.

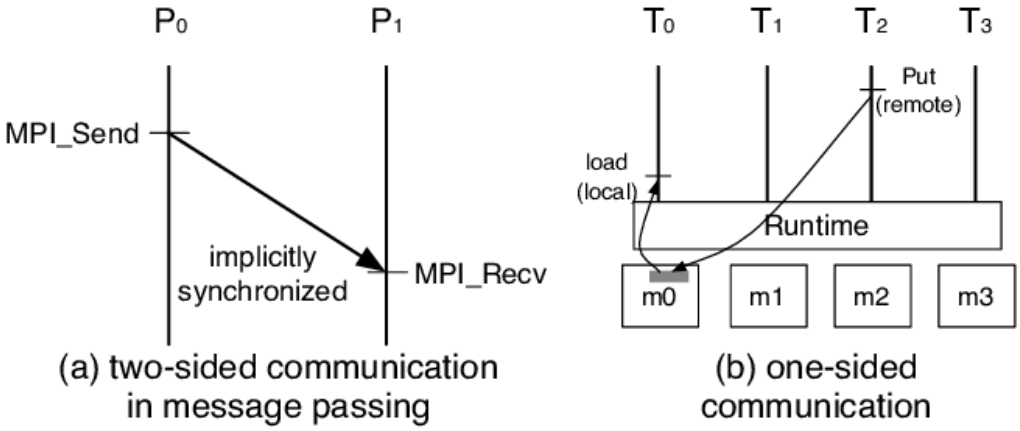
- Should be able to move data without requiring that the remote process synchronize
- Each process exposes a part of its memory to other processes
- Other processes can directly read from or write to this memory

In one-sided MPI operations, also known as **RDMA** or **RMA** (Remote Memory Access) operation.

Advantages of RMA Operations

- Can do multiple data transfers with a single synchronization operation
- Bypass tag matching
- Some irregular communication patterns can be more economically expressed

- Can be significantly faster than send/receive on systems with hardware support for remote memory access, such as shared memory systems.

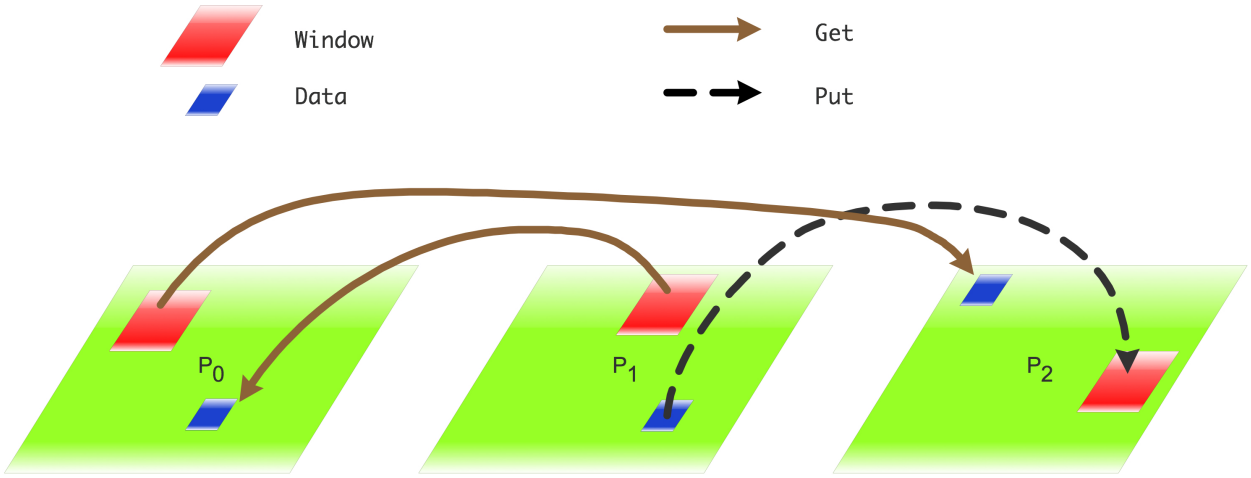


Irregular Communication Patterns with RMA

- If communication pattern is not known *a priori*, but the data locations are known, the send-receive model requires an extra step to determine how many sends-receives to issue
- RMA, however, can handle it easily because only the origin or target process needs to issue the put or get call
- This makes dynamic communication easier to code in RMA

Creating Public Memory

- Any memory created by a process is, by default, only locally accessible
- Once the memory is created, the user has to make an explicit MPI call to declare a memory region as remotely accessible
  - MPI terminology for remotely accessible memory is a"window"
  - A group of processes collectively create a "window object"
- Once a memory region is declared as remotely accessible, all processes in the window object can read/write data to this memory without explicitly synchronizing with the target process



Basic RMA Functions for Communication

- MPI_Win_create exposes local memory to RMA operation by other processes in a communicator
- Creates window object MPI_Win_free deallocates window object
- MPI_Win_Create_Dynamic creates an RMA window, to which data can later be attached.
  - Only data exposed in a window can be accessed with RMA ops
  - Initially "empty"
  - Application can dynamically attach/detach memory to this window by calling MPI_Win_attach/detach
  - Application can access data on this window only after a memory region has been attached
- MPI_Put moves data from local memory to remote memory
- MPI_Get retrieves data from remote memory into local memory
- MPI_Accumulate updates remote memory using local values

Data movement operations are **non-blocking**.

Subsequent synchronization on window object needed to ensure operation is completed.

1.7.2 Parallel I/O

I/O in HPC Applications

High Performance Computing (HPC) applications often

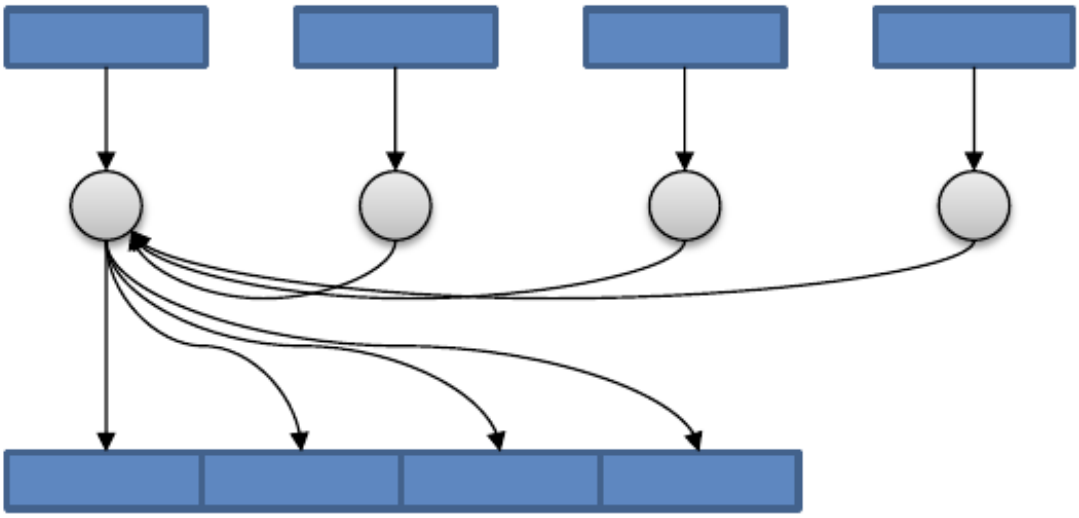
- Read initial conditions or datasets for processing
- Write numerical data from simulations
  - Saving application-level checkpoints
- In case of large distributed HPC applications, the total execution time can be broken down into **the computation time**, **communication time**, and the **I/O time**
- Optimizing the time spent in computation, communication and I/O can lead to overall improvement in the application performance
- However, doing efficient I/O without stressing out the HPC system is challenging and often an **afterthought**

Addressing the I/O Bottlenecks

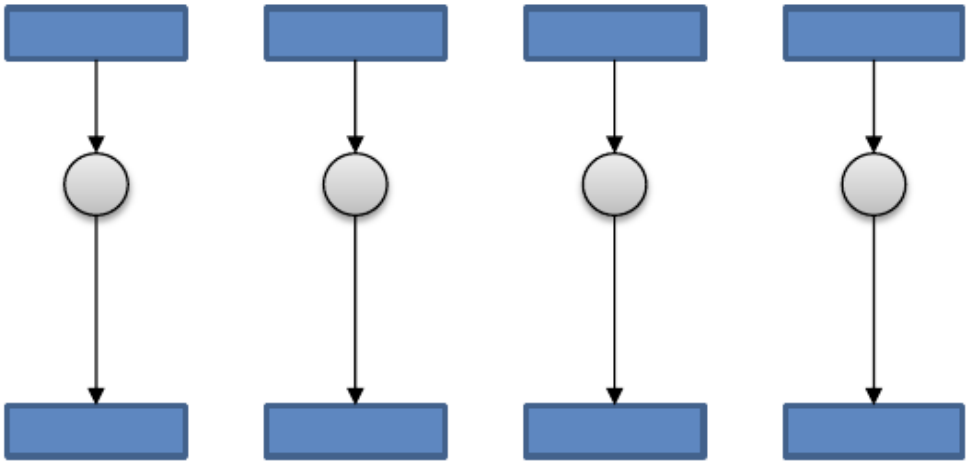
- **Software support for parallel I/O is available in the form of**
  - Parallel distributed file systems that provide parallel data paths to storage disks
  - MPI I/O
  - Libraries like PHDF5, pNetCDF
  - High-level libraries like T3PIO
- Understand the I/O strategies for maintaining good citizenship on a supercomputing resource

Real-World Scenario

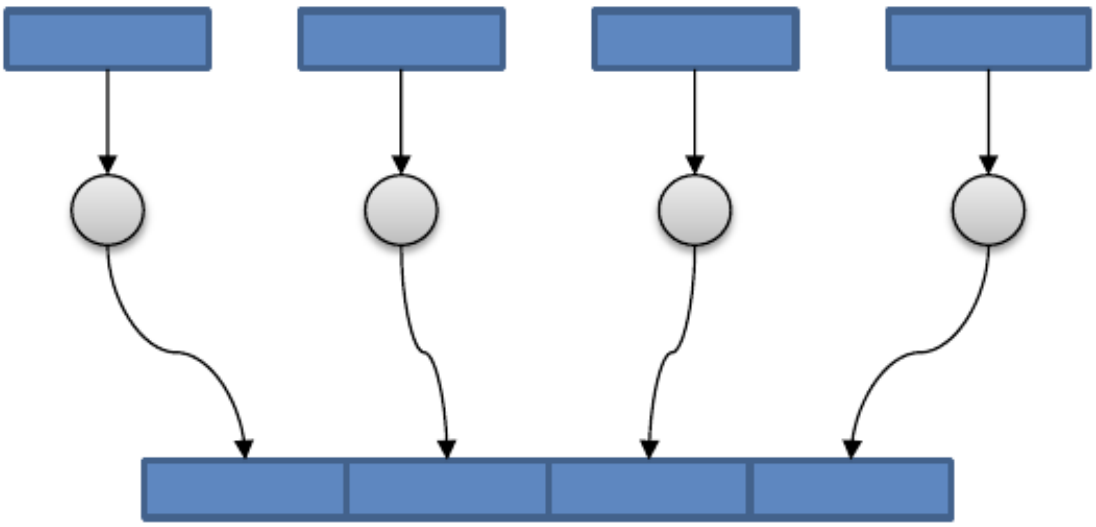
Parallel Programs Doing Sequential I/O



Parallel I/O - One file per process



Parallel I/O - Shared file (What we want)



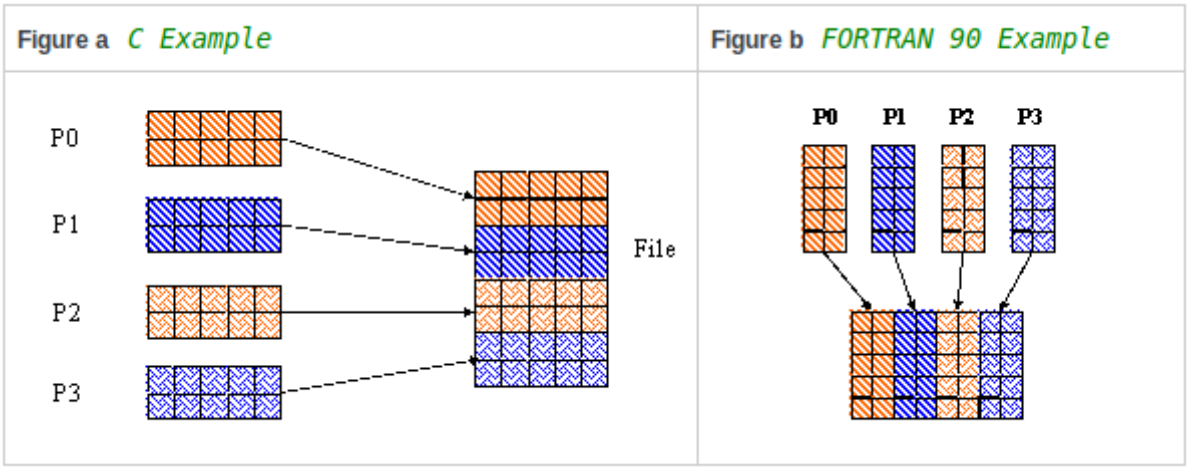
MPI I/O

- **Defined in the MPI standard since 2.0**
  - Uses MPI datatypes to describe files
  - Uses send/receive like operations to read/write data
  - Common interface for all platform/languages
- Provides high-performance (parallel) I/O operations

HDF5: Hierarchical Data Format

HDF5 Nice Features

- Interface support for C, C++, Fortran, Java, and Python
- Supported by data analysis packages (Matlab, IDL, Mathematica, Octave, Visit, Paraview, Tekplot, etc.)
- Machine independent data storage format
- Supports user defined datatypes and metadata
- Read or write to a portion of a dataset (Hyperslab)
- Runs on almost all systems



PHDF5 Overview

- **PHDF5 is the Parallel HDF5 library.**
  - You can write one file in parallel efficiently!
  - Parallel performance of HDF5 very close to MPI I/O.
- Uses MPI I/O (Don't reinvent the wheel)
- MPI I/O techniques apply to HDF5.

1.8 Profiling Method

1.8.1 Which is the Time Consuming Routine?

Use Gprof

Gprof is a performance analysis tool used to profile applications to determine where time is spent during program execution. Gprof is included with most Unix/Linux implementations, is simple to use, and can quickly show which parts of an application take the most time (hotspots). Gprof works by automatically instrumenting your code during compilation, and then sampling the application's program counter during execution. Sampling data is saved in a file, typically named gmon.out, which can then be read by the gprof command.

Typical Workflow

- compile/link with *-pg* option
- Set output file (by default gmon.out)
  - export GMON_OUT_PREFIX=<gprof_output_file>
- To see profile and callpath
  - gprof <executable> <gprof_output_file>

Example

Serial performance 4 elements, time step = 2.0e-4, domain [0.0, 1.0], polynomial order: 6

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 34.67      0.26     0.26                             __basis_MOD_matrix_vector_derivative
 14.67      0.37     0.11                             __a_times_spatial_der_MOD_a_times_spatial_derirative_y
 13.33      0.47     0.10                             __a_times_spatial_der_MOD_a_times_spatial_derirative_x
 13.33      0.57     0.10                             __spactial_derivative_MOD_dg_spatial_derivative
  4.00      0.60     0.03                             __dg_time_der_all_MOD_dg_time_der_combine
  2.67      0.62     0.02                             __basis_MOD_interpolate_to_boundary
  2.67      0.64     0.02                             __interfaces_construct_MOD_construct_interfaces_x
  2.67      0.66     0.02                             __numerical_flux_MOD_numerical_flux_y
  2.67      0.68     0.02                             __numerical_flux_MOD_riemann1
  1.33      0.69     0.01                             __external_state_MOD_external_state_gaussian_exact
  1.33      0.70     0.01                             __interfaces_construct_MOD_construct_interfaces_y
  1.33      0.71     0.01                             __numerical_flux_MOD_numerical_flux_x
  1.33      0.72     0.01                             __riemann_solver_MOD_riemann_y
  1.33      0.73     0.01                             __time_step_by_rk_MOD_dg_step_by_rk3
  0.67      0.74     0.01                             __flux_vectors_MOD_xflux
  0.67      0.74     0.01                             __flux_vectors_MOD_yflux
  0.67      0.75     0.01                             __poly_level_and_order_MOD_poly_level_to_order
  0.67      0.75     0.01                             __poly_level_and_order_MOD_poly_order_to_level
```

Parallel MPI One-sided Communication

2 processors

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 22.08      0.27     0.27  5040000     0.05     0.05  __basis_MOD_matrix_vector_derivative
 13.33      0.43     0.16  1680000     0.10     0.25  __spactial_derivative_MOD_dg_spatial_derivative
 11.67      0.57     0.14    60000     2.33    19.00  __dg_time_der_all_MOD_dg_time_der_combine
 10.83      0.70     0.13   120000     1.08     2.90  __a_times_spatial_der_MOD_a_times_spatial_derirative_x
  9.17      0.81     0.11   120000     0.92     2.77  __a_times_spatial_der_MOD_a_times_spatial_derirative_y
  7.50      0.90     0.09   120000     0.75     1.25  __numerical_flux_MOD_numerical_flux_y
  5.00      0.96     0.06   120000     0.50     0.79  __numerical_flux_MOD_numerical_flux_x
  4.58      1.01     0.06 10080384     0.01     0.01  __basis_MOD_interpolate_to_boundary
  4.17      1.06     0.05    20000     2.50    59.50  __time_step_by_rk_MOD_dg_step_by_rk3
  2.92      1.10     0.04    60000     0.58     0.64  __numerical_flux_MOD_riemann2
  1.67      1.12     0.02   240004     0.08     0.08  __hilbert_MOD_d2xy
  1.67      1.14     0.02    60000     0.33     0.33  __numerical_flux_MOD_riemann1
  0.83      1.15     0.01  5880000     0.00     0.00  __flux_vectors_MOD_yflux
  0.83      1.16     0.01  1680000     0.01     0.01  __external_state_MOD_external_state_gaussian_exact
  0.83      1.17     0.01  1260000     0.01     0.01  __riemann_solver_MOD_riemann_y
  0.83      1.18     0.01   120004     0.08     0.31  __interfaces_construct_MOD_construct_interfaces_x
  0.83      1.19     0.01   120004     0.08     0.31  __interfaces_construct_MOD_construct_interfaces_y
  0.42      1.19     0.01  5880000     0.00     0.00  __flux_vectors_MOD_xflux
  0.42      1.20     0.01                             __basis_MOD_almostequal
  0.42      1.20     0.01                             __basis_MOD_legendre_polynomial_and_derivative
  0.00      1.20     0.00  1680728     0.00     0.00  __affine_map_MOD_affine_mapping
  0.00      1.20     0.00  1680028     0.00     0.00  __poly_level_and_order_MOD_poly_level_to_order
  0.00      1.20     0.00  1260000     0.00     0.00  __riemann_solver_MOD_riemann_x
  0.00      1.20     0.00   240000     0.00     0.00  __index_local_global_MOD_index_local_to_global
  0.00      1.20     0.00   240000     0.00     0.00  __mpi_boundary_MOD_create_window
  0.00      1.20     0.00   120012     0.00     0.00  __hilbert_MOD_xy2d
  0.00      1.20     0.00   120000     0.00     0.00  __index_local_global_MOD_index_global_to_local
  0.00      1.20     0.00    60000     0.00     0.00  __search_rank_MOD_find_rank
  0.00      1.20     0.00        8     0.00     0.00  __basis_MOD_lagrange_interpolating_polynomial
  0.00      1.20     0.00        8     0.00     0.03  __output_MOD_interpolate_to_corner
  0.00      1.20     0.00        8     0.00     0.03  __output_MOD_x_interface
  0.00      1.20     0.00        4     0.00     0.00  __basis_MOD_barw
```

Parallel MPI Non-blocking Communication

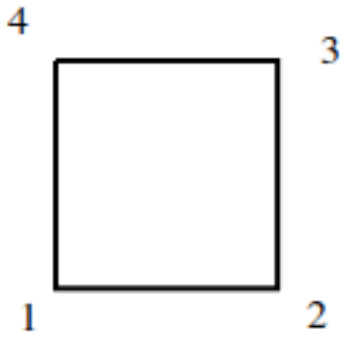
2 processors

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 22.45      0.11     0.11   120000     0.92     1.42  __a_times_spatial_der_MOD_a_times_spatial_derirative_x
 14.29      0.18     0.07  1200000     0.06     0.10  __spactial_derivative_MOD_dg_spatial_derivative
 10.20      0.23     0.05  3600000     0.01     0.01  __basis_MOD_matrix_vector_derivative
  8.16      0.27     0.04  7200288     0.01     0.01  __basis_MOD_interpolate_to_boundary
  8.16      0.31     0.04   120000     0.33     0.83  __a_times_spatial_der_MOD_a_times_spatial_derirative_y
  6.12      0.34     0.03    60000     0.50     7.83  __dg_time_der_all_MOD_dg_time_der_combine
  4.08      0.36     0.02   120004     0.17     0.33  __interfaces_construct_MOD_construct_interfaces_y
  4.08      0.38     0.02    20000     1.00    24.50  __time_step_by_rk_MOD_dg_step_by_rk3
  2.04      0.39     0.01  1200400     0.01     0.01  __affine_map_MOD_affine_mapping
  2.04      0.40     0.01  1200000     0.01     0.01  __external_state_MOD_external_state_gaussian_exact
  2.04      0.41     0.01   900000     0.01     0.01  __riemann_solver_MOD_riemann_x
  2.04      0.42     0.01   120004     0.08     0.25  __interfaces_construct_MOD_construct_interfaces_x
  2.04      0.43     0.01   120000     0.08     0.33  __numerical_flux_MOD_numerical_flux_x
  2.04      0.44     0.01    60000     0.17     0.17  __message_exchage_MOD_exchange_nflux_x
  2.04      0.45     0.01    60000     0.17     0.17  __message_exchage_MOD_exchange_nflux_y
  2.04      0.46     0.01    60000     0.17     0.17  __message_exchage_MOD_exchange_solution_x
  2.04      0.47     0.01    60000     0.17     0.17  __message_exchage_MOD_exchange_solution_y
  2.04      0.48     0.01    60000     0.17     0.22  __numerical_flux_MOD_riemann1
  2.04      0.49     0.01    60000     0.17     0.17  __numerical_flux_MOD_riemann2
  0.00      0.49     0.00  3000000     0.00     0.00  __flux_vectors_MOD_xflux
  0.00      0.49     0.00  3000000     0.00     0.00  __flux_vectors_MOD_yflux
  0.00      0.49     0.00  1680028     0.00     0.00  __poly_level_and_order_MOD_poly_level_to_order
  0.00      0.49     0.00   900000     0.00     0.00  __riemann_solver_MOD_riemann_y
  0.00      0.49     0.00   480000     0.00     0.00  __index_local_global_MOD_index_local_to_global
  0.00      0.49     0.00   360004     0.00     0.00  __hilbert_MOD_d2xy
  0.00      0.49     0.00   240012     0.00     0.00  __hilbert_MOD_xy2d
  0.00      0.49     0.00   120000     0.00     0.00  __message_exchage_MOD_non_blocking_excahnge_interface_x
  0.00      0.49     0.00   120000     0.00     0.00  __message_exchage_MOD_non_blocking_excahnge_nflux_y
  0.00      0.49     0.00   120000     0.00     0.17  __numerical_flux_MOD_numerical_flux_y
  0.00      0.49     0.00   120000     0.00     0.00  __search_rank_MOD_find_rank
  0.00      0.49     0.00    60000     0.00     0.00  __index_local_global_MOD_index_global_to_local
  0.00      0.49     0.00        8     0.00     0.00  __basis_MOD_lagrange_interpolating_polynomial
```

1.9 Some features in the solver

1.9.1 Element Node-ordering Format

Element node-ordering follows the format shown below:



1.9.2 Data Storage

Element coordinates

For a quadrilateral mesh, only the coordinates of two diagonal modes are stored. To be specific, point 1 and point 3.

1.10 Reference

1. Kopriva, David. (2009). Implementing Spectral Methods for Partial Differential Equations.

2.Toro, Eleuterio. (2009). Riemann Solvers and Numerical Methods for Fluid Dynamics.

3.Hesthaven, Jan & Warburton, Tim. (2008). Nodal Discontinuous Galerkin Method.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[HKR+12] Daniel F. Harlacher, Harald Klimach, Sabine Roller, Christian Siebert, and Felix Wolf. Dynamic load balancing for unstructured meshes on space-filling curves. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1661–1669, 2012.

[TSA06] Srikanta Tirthapura, Sudip Seal, and Srinivas Aluru. A formal analysis of space filling curves for parallel domain decomposition. *2006 International Conference on Parallel Processing (ICPP'06)*, pages 505–512, 2006.

[ZBMZ+18] Keke Zhai, Tania Banerjee-Mishra, David Zwick, Jason Hackl, and Sanjay Ranka. Dynamic load balancing for compressible multiphase turbulence. *ArXiv*, 2018.