Scientific Computing & HPC

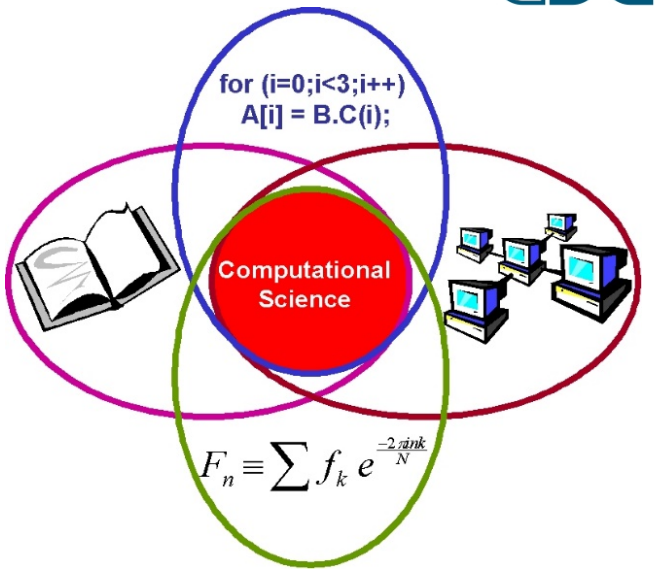
A brief Introduction

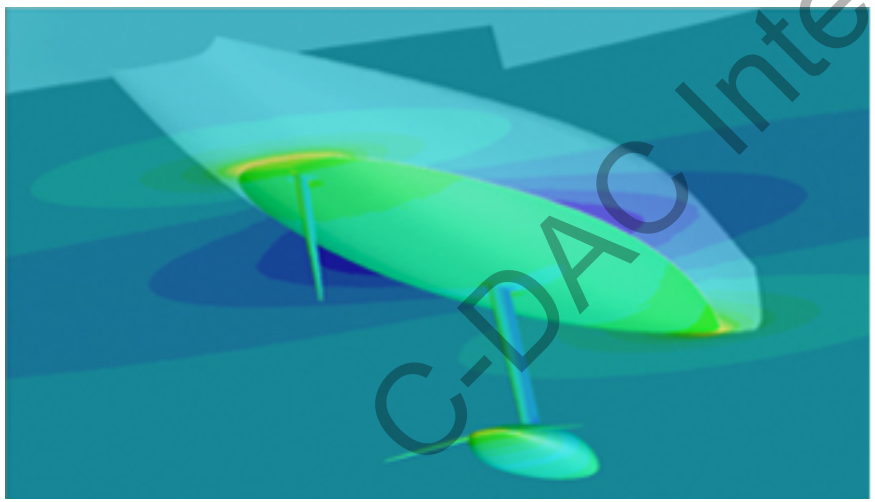
CENTER FOR DEVELOPMENT OF
ADVANCED COMPUTING

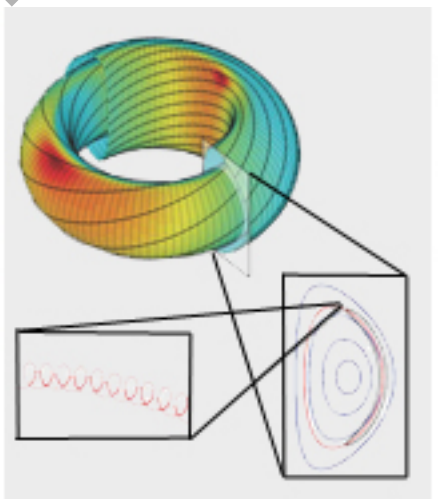
Waqqas Zia
HPC-Technologies Group
CDAC-Pune

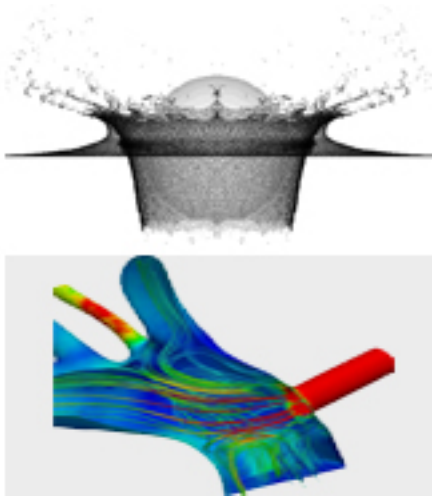
What is Scientific Computing ?

Scientific computing is the collection of tools, techniques, and theories required to solve on a computer, mathematical models of problems in science and engineering.



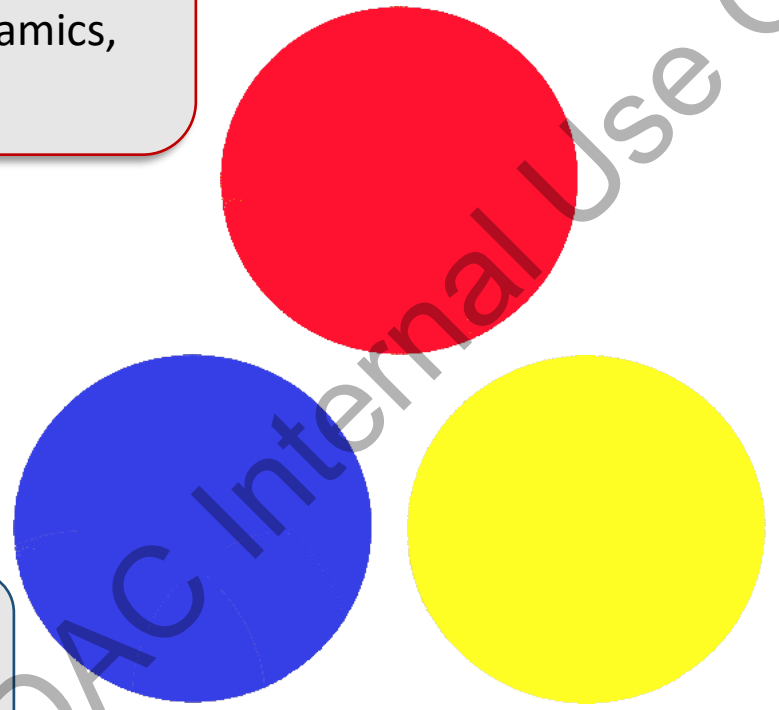






Science & Engineering

Astrophysics, quantum chemistry, Fluid dynamics, Bio-informatics etc..

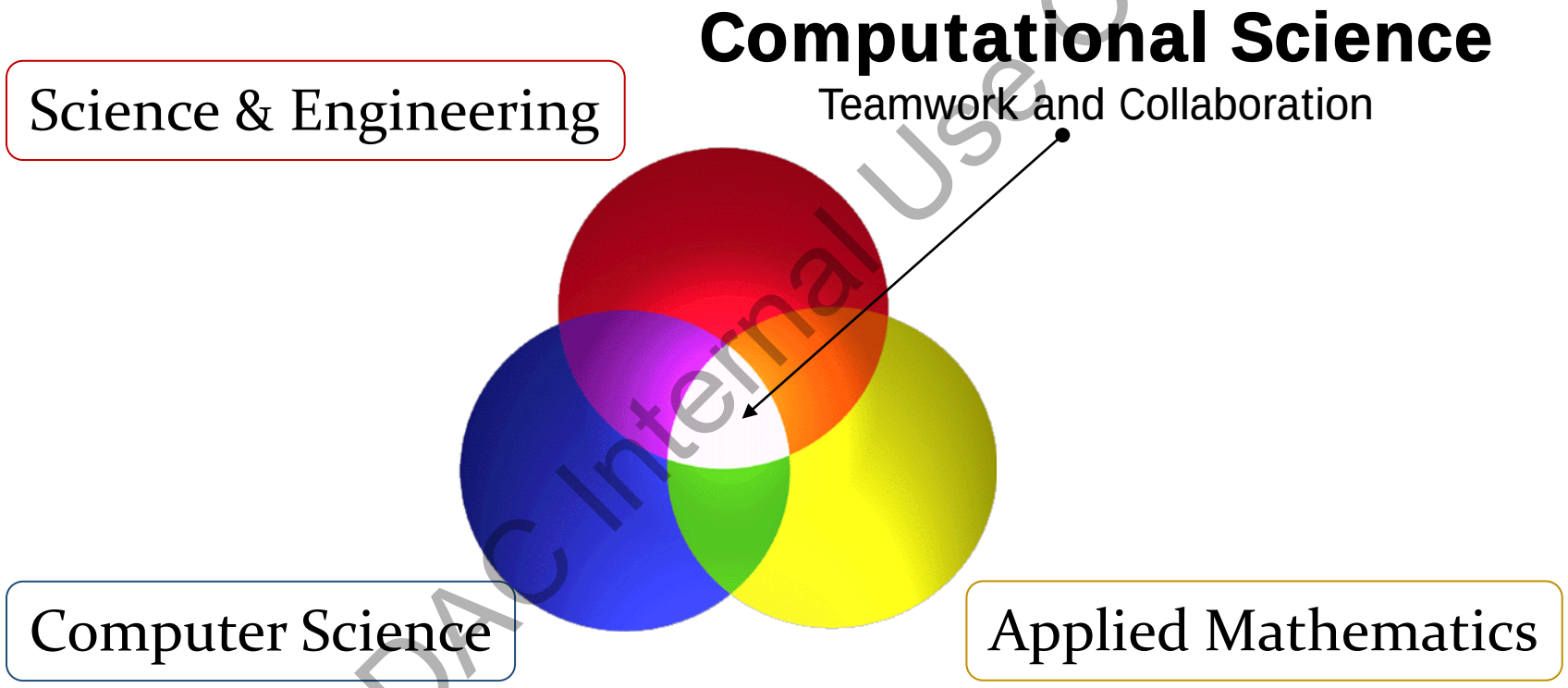


Computer Science

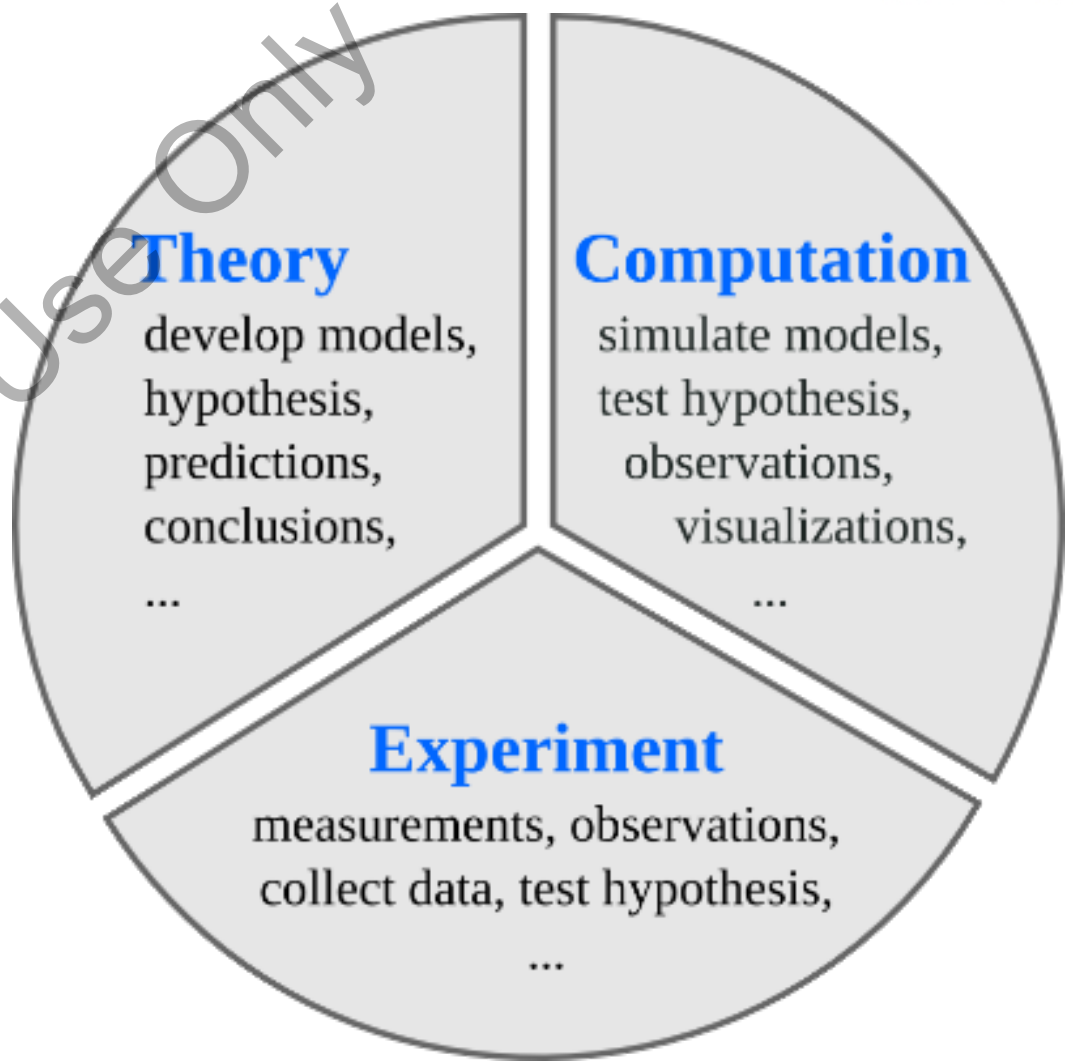
Application software/ Hardware

Applied Mathematics

Numerical Analysis, mathematical Modelling, Simulations etc.



- Complements, but does not replace, theory and experimentation in scientific research.
- Third branch of understanding Science



What's its significance??

The solution or simulation of fun
engineering, with a strong scientif
Grand Challenge Problems (GCPs), h
Parallel Computing.

Large volumes of data are produced and their processing and analysis also require computing

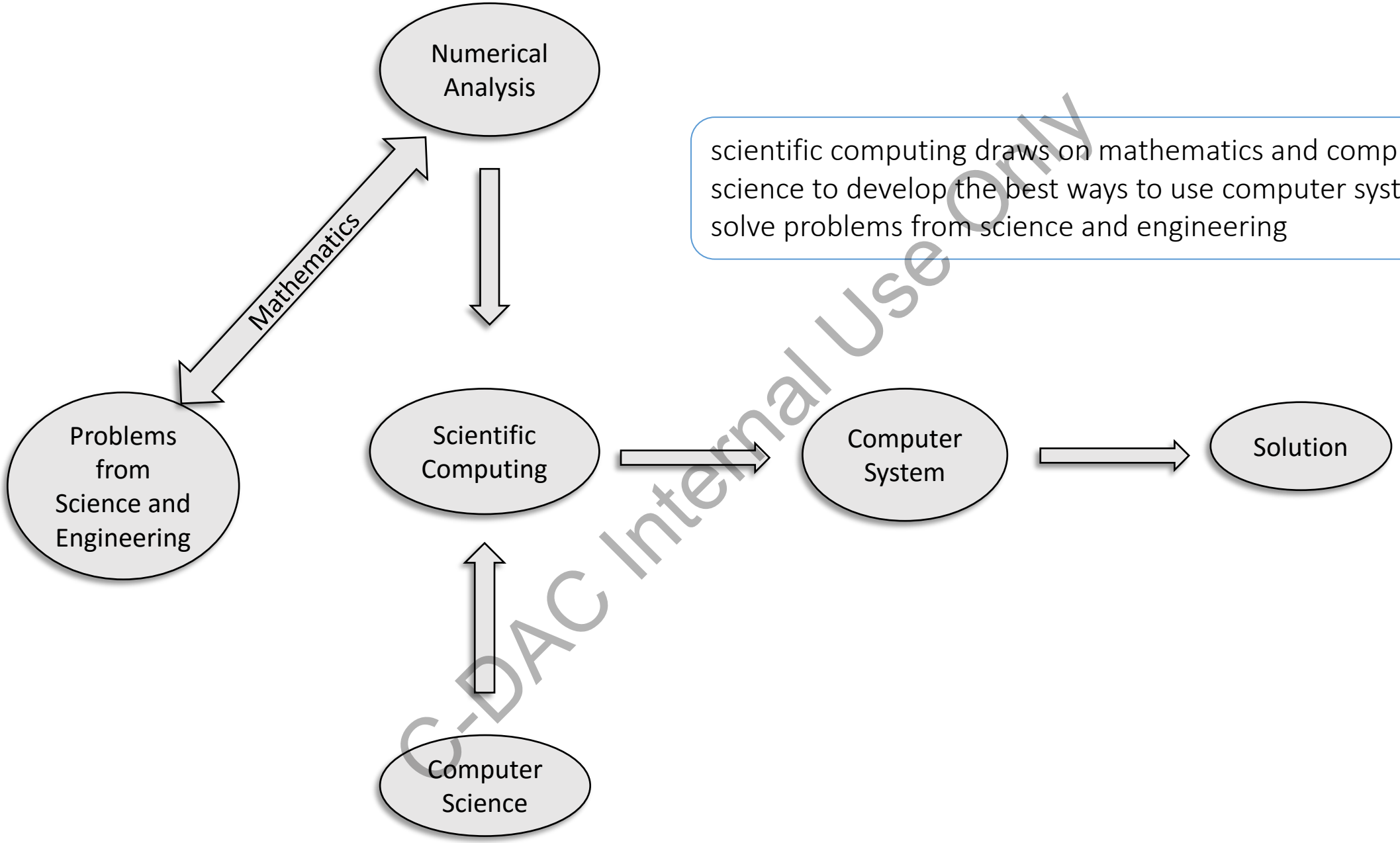
- Data mining
- Web search
- networked video
- Video games and virtual reality
- Computer aided medical diagnosis

- Global Climate modeling
- Biology: genomics; protein f
- Astrophysical modeling
- Computational Chemistry

Data is collected and stored at enormous speeds (GByte/hour)

- C
- C
- I

- Sensor data streams
- Telescope scanning the skies
- Micro-arrays generating gene expression data



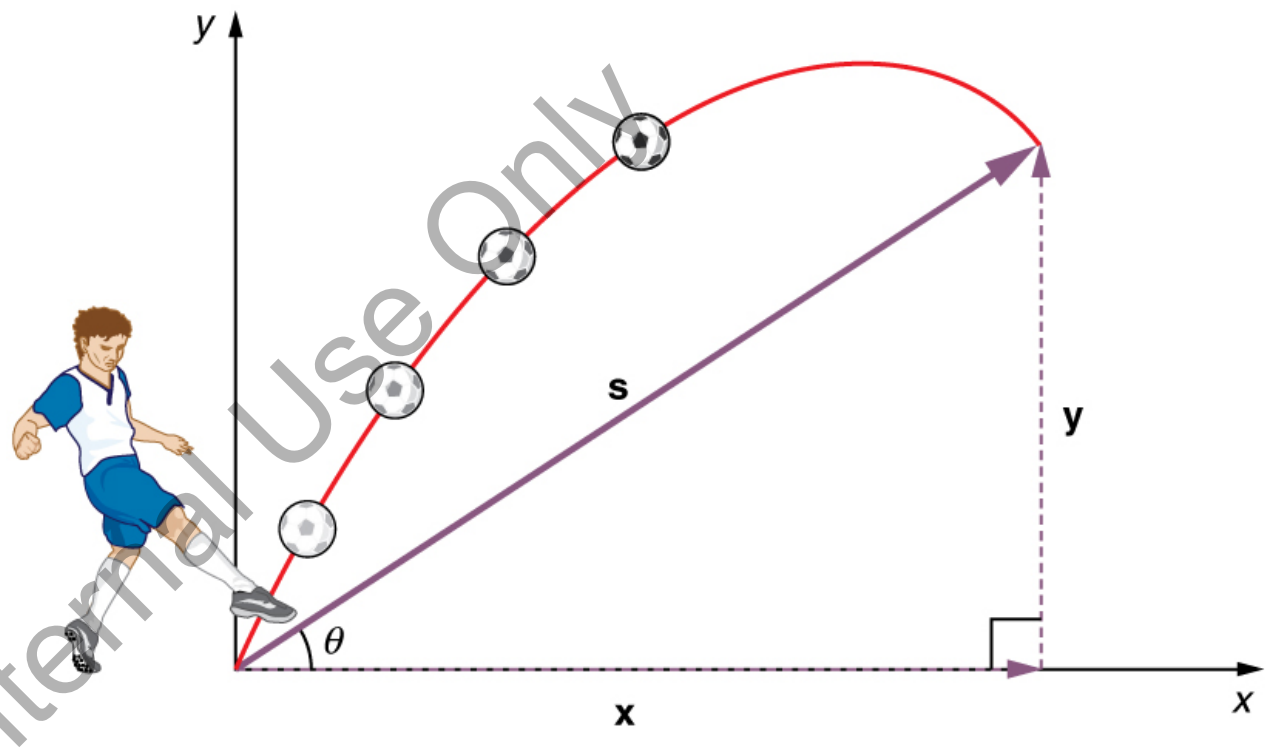
scientific computing draws on mathematics and computer science to develop the best ways to use computer systems to solve problems from science and engineering

First Step??

Mathematical Modelling

- Modelling
- Validation

Modelling
Projectile motion



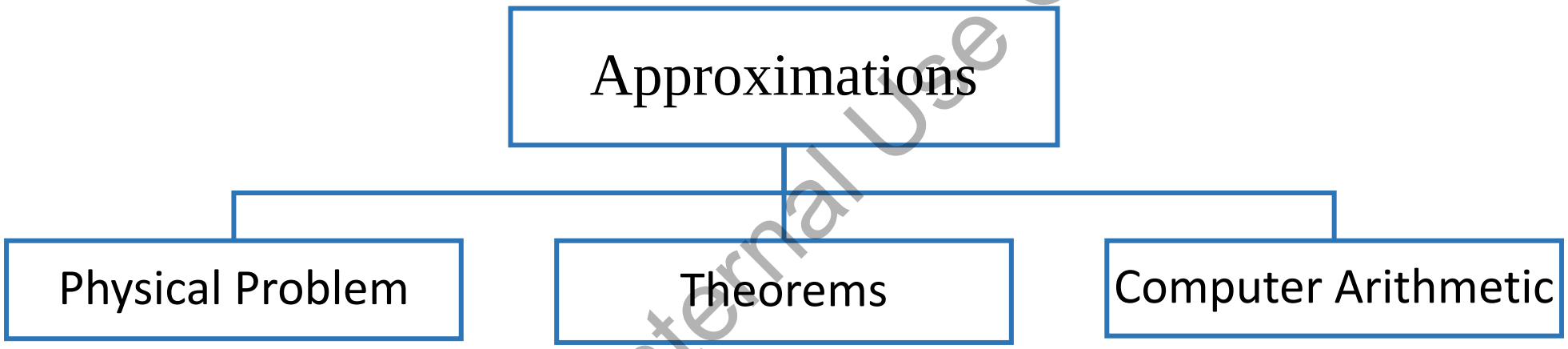
Simple two Dimensional Kinematics Problem

$$v_y = v_{0y} - gt \qquad y = y_0 + v_{0y}t - \frac{1}{2}gt^2$$

$$y = y_0 + \frac{1}{2}(v_{0y} + v_y)\,t \qquad v_y^2 = v_{0y}^2 - 2g(y - y_0)$$

Other physical problems are similarly modeled by enumerating and quantifying the forces that will be of importance

Approximations are made



Computer Arithmetic

- Related to machine precision
- Round off or truncation error
- only a small number of real numbers can be represented exactly

Machine precision ($\boldsymbol{\varepsilon}$) can be defined as: $\boldsymbol{\varepsilon}$ is the smallest number that can be added to **1** so that **1** + $\boldsymbol{\varepsilon}$ has a different representation than **1**

$$\begin{array}{rl} & 1.0000 \times 10^{0} \\ + & 1.0000 \times 10^{-5} \end{array} \Longrightarrow \begin{array}{rl} & 1.0000 \times 10^{0} \\ + & 0.00001 \times 10^{0} \\ = & 1.0000 \times 10^{0} \end{array}$$

- aligning exponents can shift a too small operand so that it is effectively ignored in the addition operation
- in the addition *x*+*y*, if the ratio of *x* and *y* is too large, the result will be identical to *x*

Summing Series

$$\sum_{n=1}^{10000} \frac{1}{n^2} = 1.644834$$

- Working with single precision (10^{-7})
- the ratio between terms, and the ratio of terms to partial sums, is ever increasing
- the last 7000 terms are ignored
- Computed sum is 1.644725. The first 4 digits are correct

- Evaluate the sum in reverse order we obtain the exact result in single precision
- the ratio will never be as bad as 10^7

- Series that are monotone (or close to monotone) should be summed from small to large
- Error is minimized if the quantities to be added are closer in magnitude
- Addition in computer arithmetic is not *associative*

Unstable algorithms

Recurrence Relation:

$$y_n = \int_0^1 \frac{x^n}{x+5}\,dx = \frac{1}{n} - 5y_{n-1}$$

$$y_0 = \ln 6 - \ln 5$$

Performing the computation in 3 decimal digits

Computation		Correct result
y_0 = 0.182		y_0 = 0.182
y_1 = 0.0900		y_1 = 0.0884
y_2 = 0.0500		y_2 = 0.0580
y_3 = 0.0830	going up??	y_3 = 0.0431
y_4 = -0.165	negative??	y_4 = 0.0343

Computed results are quickly not just inaccurate, but actually nonsensical

Error Analysis

Error $\boldsymbol{\varepsilon_n}$ in the $\boldsymbol{n}$-th step,

$$y_n - \widetilde{y_n} = \varepsilon_n$$

Then,

$$\widetilde{y_n} = \frac{1}{n} - 5\widetilde{y_{n-1}} = \frac{1}{n} - 5y_{n-1} + 5\varepsilon_{n-1}$$

$$\widetilde{y_n} = y_n + 5\varepsilon_{n-1}$$

So ,

$$\varepsilon_n \geq 5\varepsilon_{n-1}$$

- Error made by this computation shows **exponential** growth

For Realistic models - Basic relations are usually PDE

THE GRAND CHALLENGE EQUATIONS

$$B_i A_i = E_i A_i + \rho_i \sum_j B_j A_j F_{ji}$$

$$dU = \left(\frac{\partial U}{\partial S}\right)_V dS + \left(\frac{\partial U}{\partial V}\right)_S dV$$

$$\sum_{k=0}^{N-1} \frac{\nabla^2 u}{\partial u}$$

$$I_{n+1} = r\, p_n (1 - p_n)$$

$$-\frac{h^2}{8\pi^2 m}\nabla^2 \Psi(r,t) + V\,\Psi(r,t) = -\frac{h}{2\pi i}\frac{\partial \Psi(r,t)}{\partial t}$$

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\vec{u} = -\frac{1}{\rho}\nabla p + \gamma \nabla^2 \vec{u} + \frac{1}{\rho}\vec{F}$$

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t}$$

$$\nabla \cdot \vec{D} = \rho$$

$$\nabla \times \vec{H} = \frac{\partial \vec{D}}{\partial t} + \vec{J}$$

$$\nabla \cdot \vec{B} = 0$$

$$\vec{F} = m\,\vec{a} + \frac{dm}{dt}\vec{v}$$

$$Z = \sum_j g_j\, e^{-E_j/kT}$$

$$P(t) = \frac{\sum_i W_i\, B_i(t)\, P_i}{\sum_i W_i\, B_i(t)}$$

$$-\nabla^2 u + \lambda\, u = f$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f$$

Apply Initial or Boundary conditions

Find a solution

• NEWTON'S EQUATIONS • SCHROEDINGER EQUATION (TIME DEPENDENT) • NAVIER-STOKES EQUATION •
• POISSON EQUATION • HEAT EQUATION • HELMHOLTZ EQUATION • DISCRETE FOURIER TRANSFORM •
• MAXWELL'S EQUATIONS • PARTITION FUNCTION • POPULATION DYNAMICS •
• COMBINED 1ST AND 2ND LAWS OF THERMODYNAMICS • RADIOSITY • RATIONAL B-SPLINE •

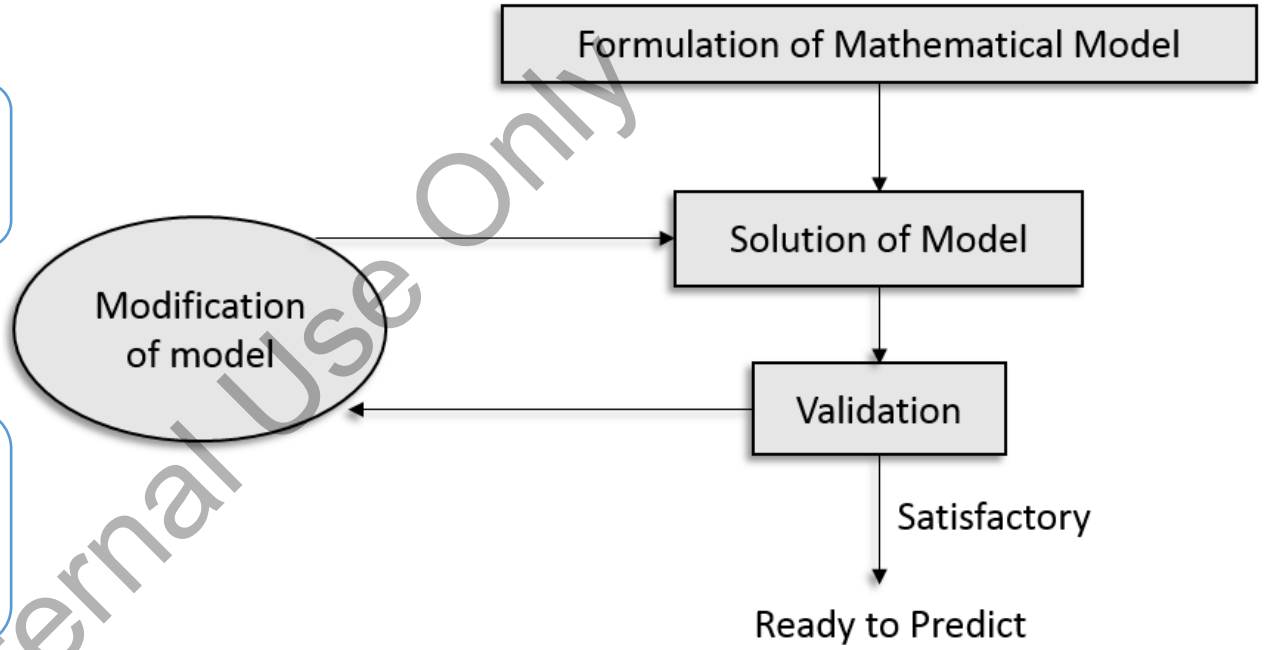
Validation of the Model

Whether the solution satisfies obvious physical and mathematical constraints

Comparison of the computed results with whatever experimental or observational data are available

X

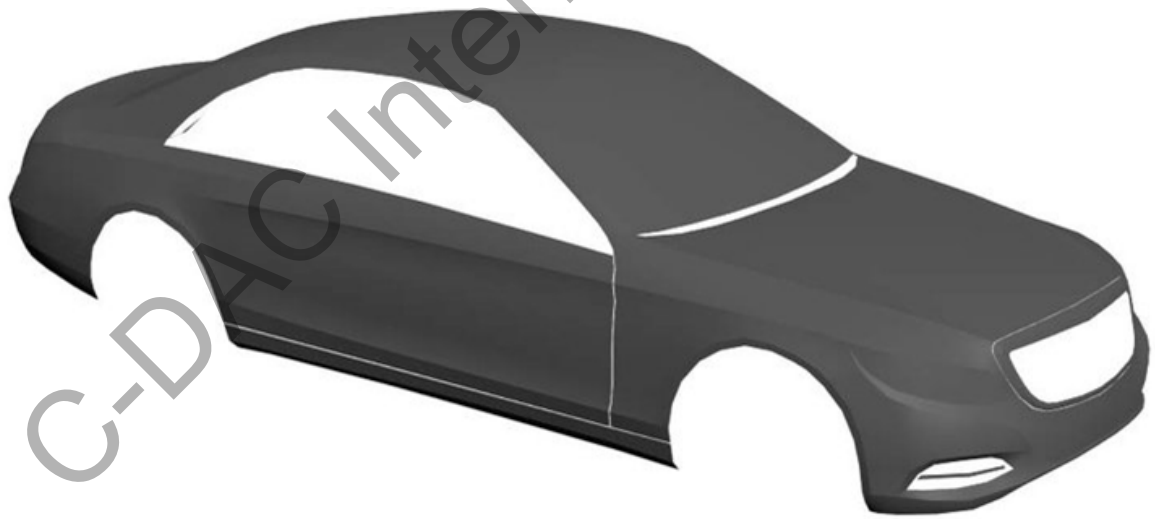
Modification of model



Experimental results may have been obtained in a controlled setting, the physics of the experiment may differ from the mathematical model

Geometry Modelling

- Computer representation of the problem domain
- Not desirable to include all the geometric details of the system
- Analyst has to choose the level of intricate details to be chosen

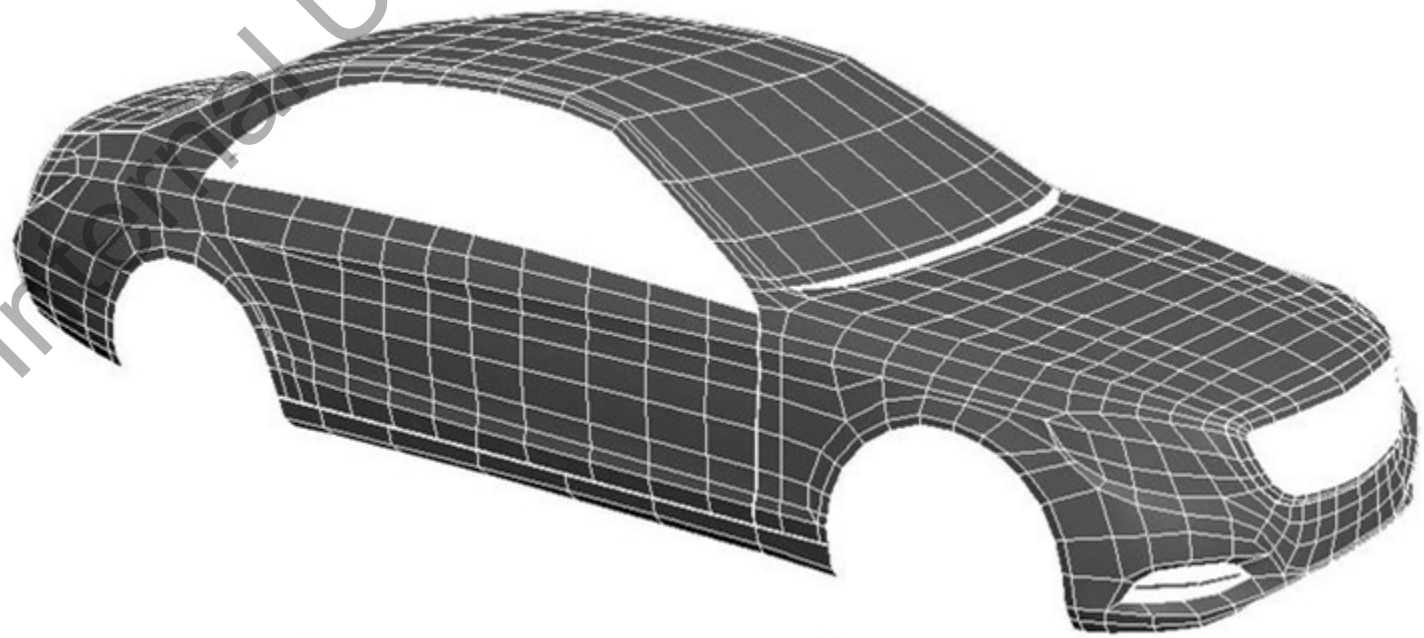


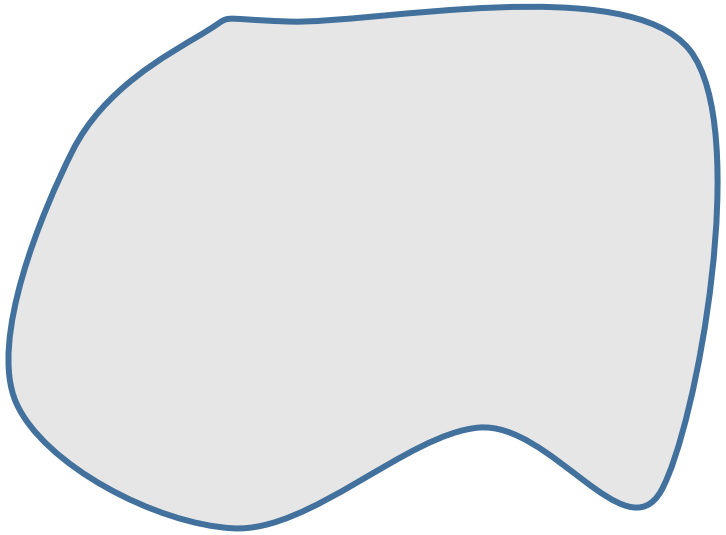
Discretization & Grid generation

Continuum mathematical model must be converted into a discrete system of algebraic equation

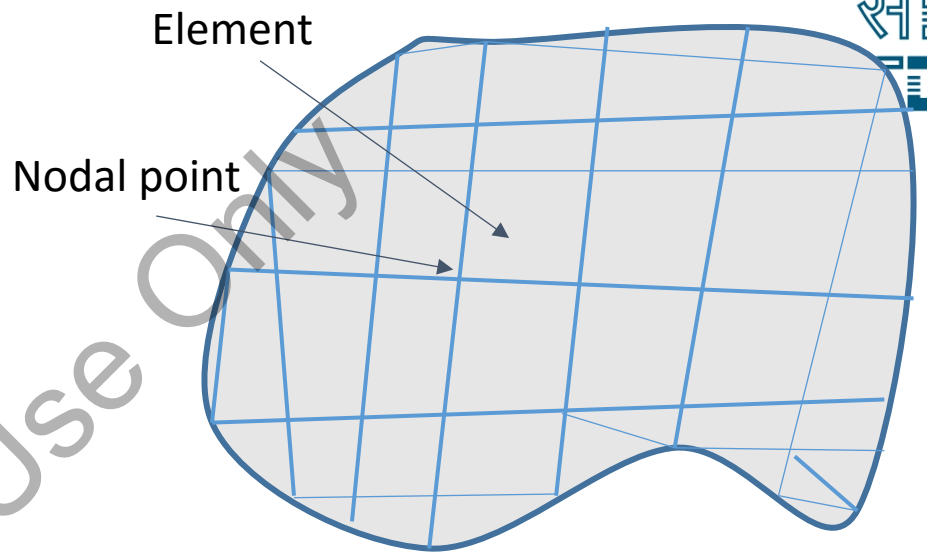
Discretization approaches

- Finite Difference Method (FDM)
- Finite Element Method (FEM)
- Finite Volume Method (FDM)





Discretization



Governing P.D.E (strong form)

$$\nabla^2 \emptyset = 0$$

Variational (weak) form

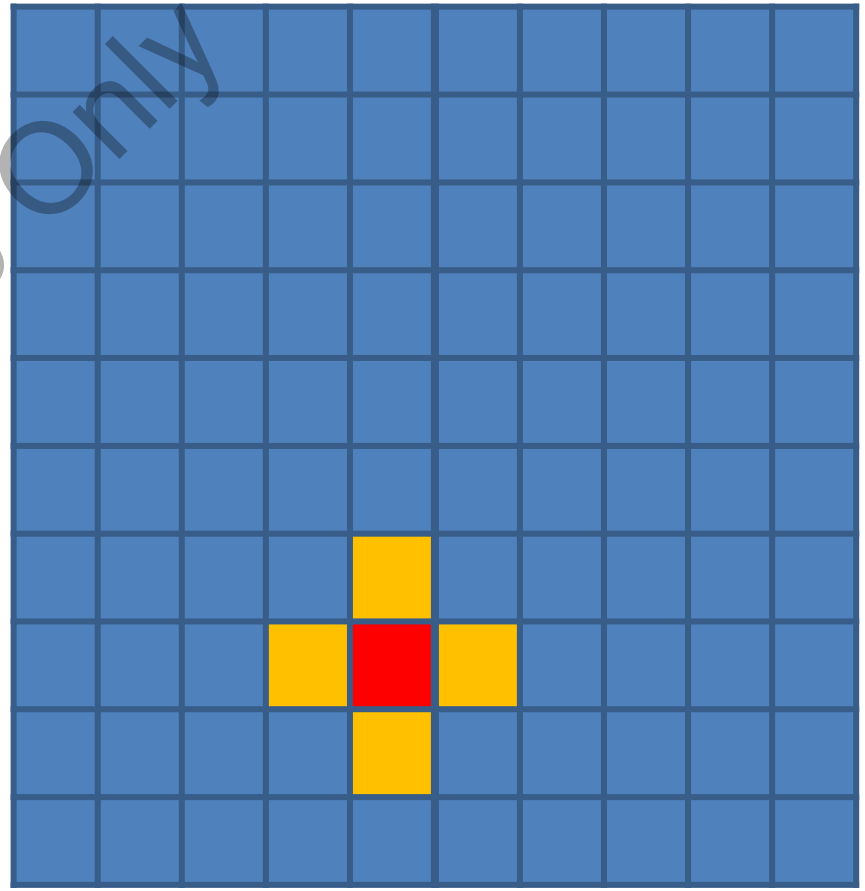
$$\int F(\emptyset, \emptyset_x, \emptyset_y, \ldots.)dx\,dy$$

Linear equations

$$A\,x = b$$

Laplace Equation

Discretization



$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

→ *Finite difference methods* →

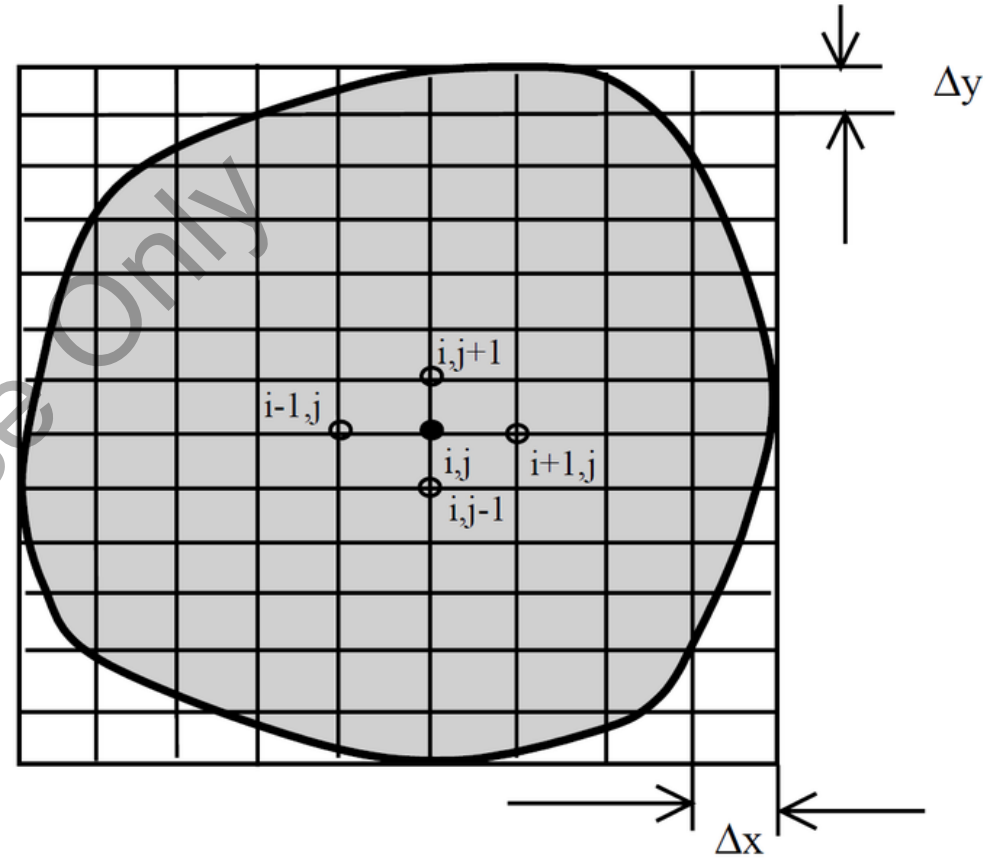
$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i.j} = 0$$

Finite Difference Method

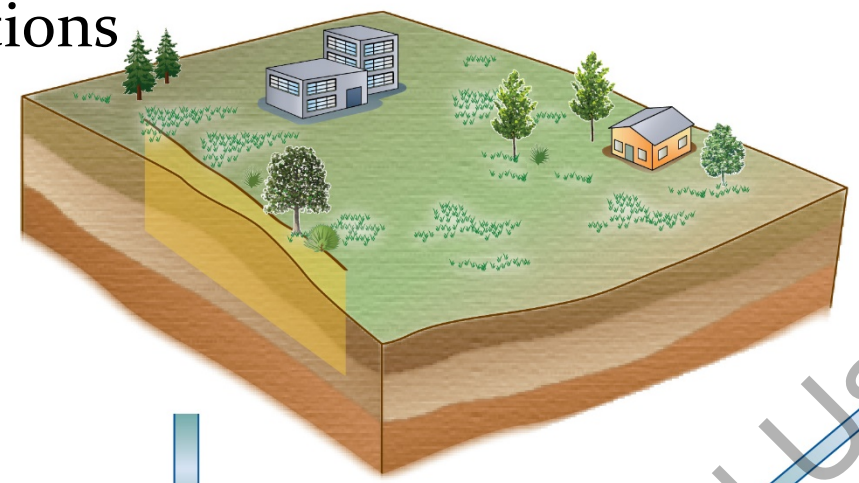
$$f(x+h) = f(x) + hf'(x) + O(h^2)$$

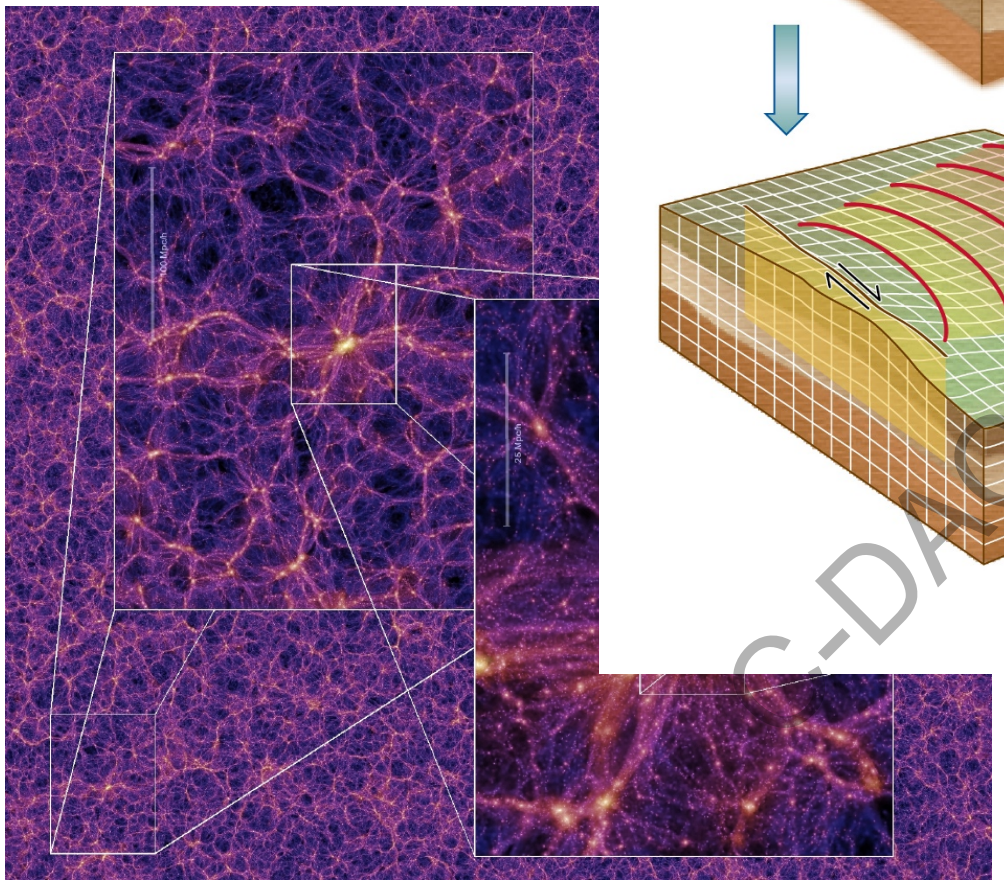
$$\frac{df}{dx} = \frac{f(x+h) - f(x)}{h}$$

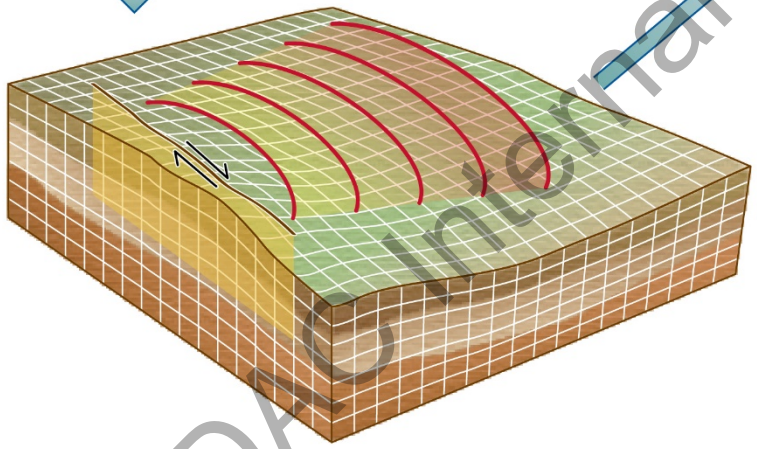
- Oldest numerical methods for PDE
- generally based on a Taylor series representation of a function
- Appropriate for structured grids only
- accuracy depends on the order of approximation
- Application areas: meteorological, seismological, and astrophysical simulations

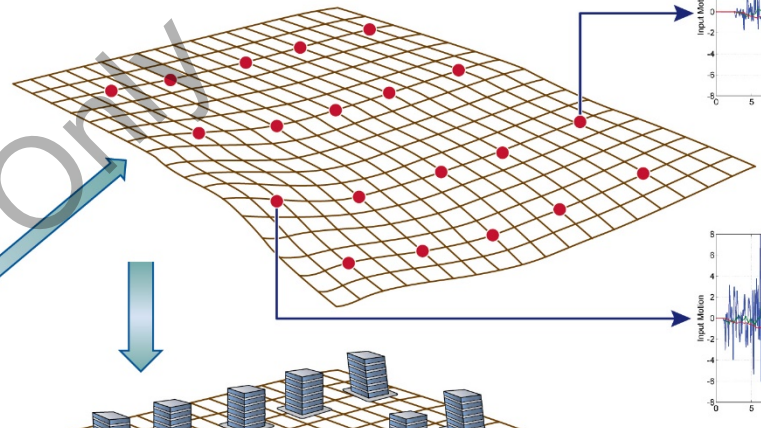


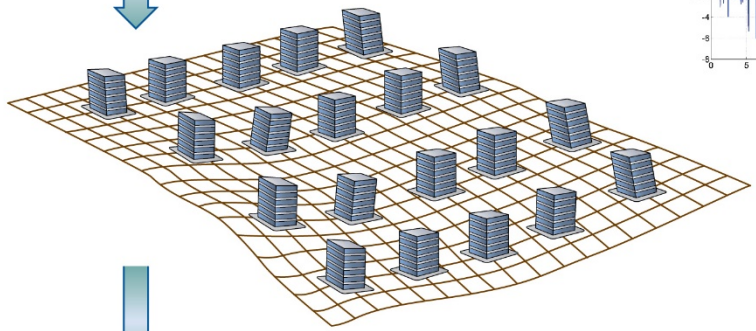
- Weather calculations
- Astrophysics
- Seismology

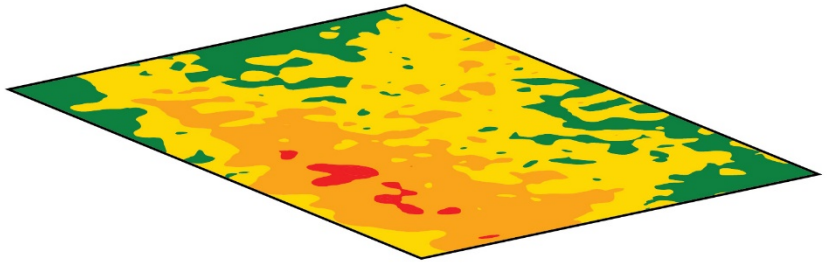






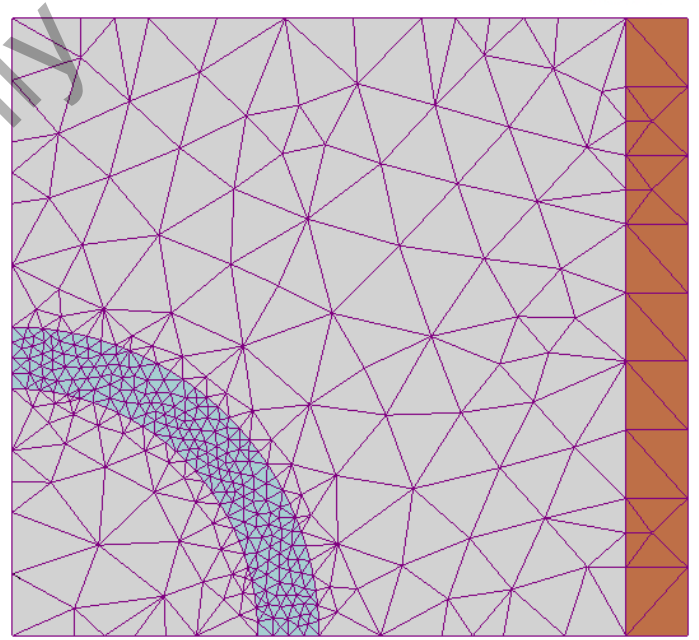




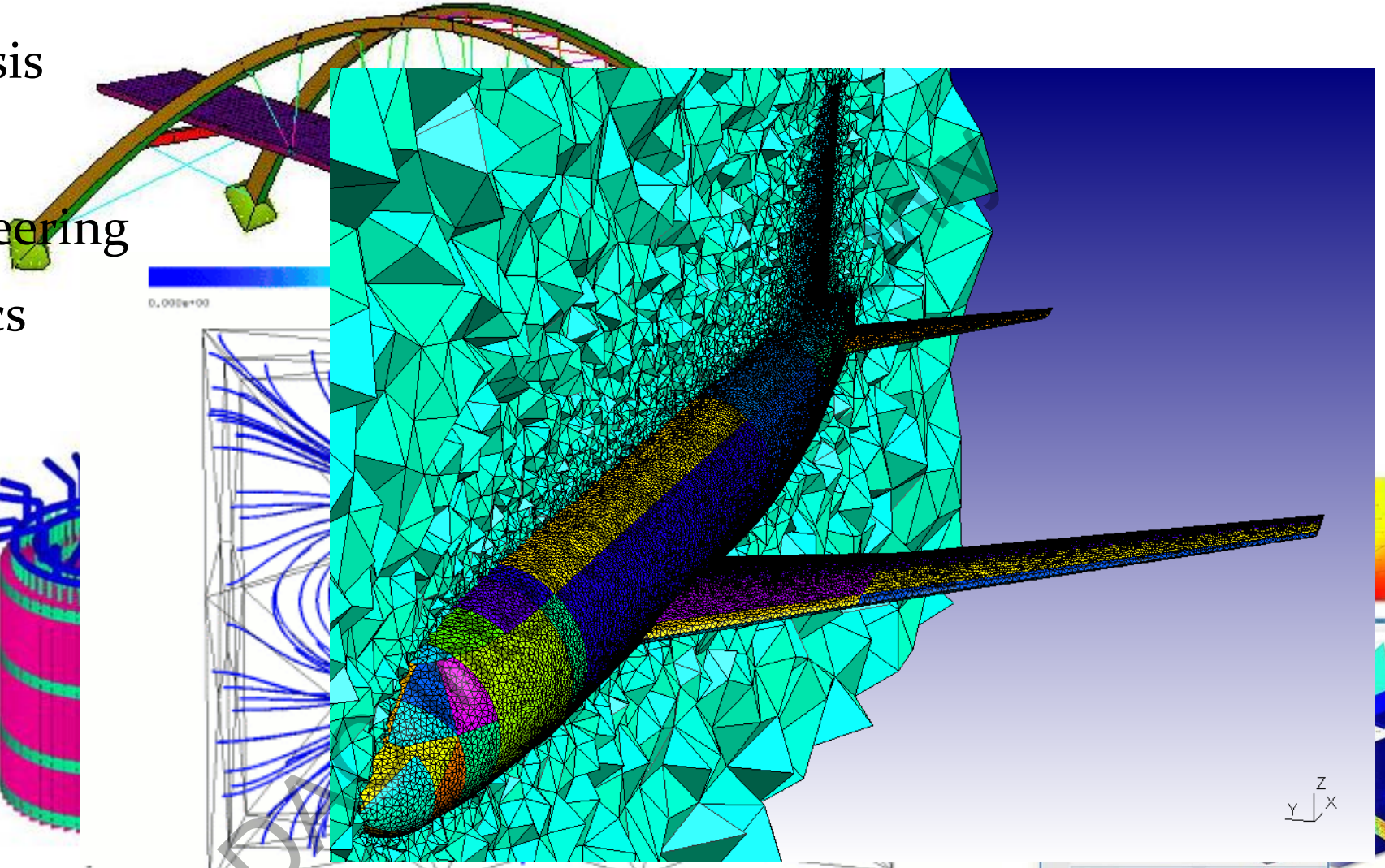


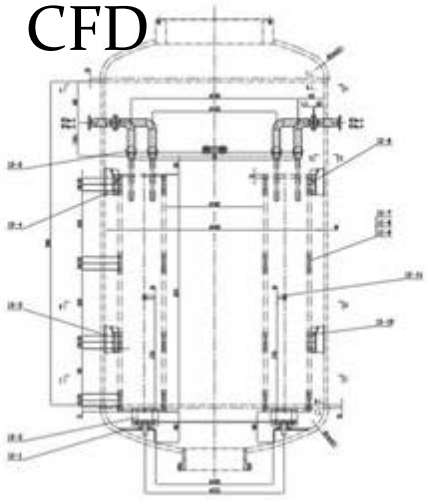
Finite Element method

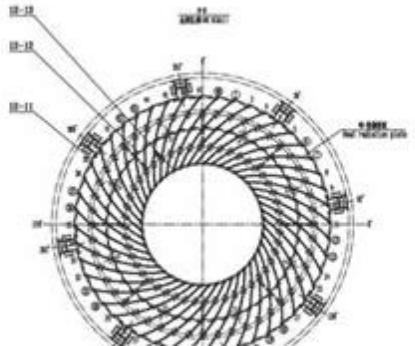
- Subdivides model into very small but finite-sized elements of geometrically simple shapes
- curved and irregular geometries are handled in a natural way
- formulate equations for each element as simple function, such as linear or quadratic polynomial
- requires quite sophisticated mathematics for its formulation
- ends up with a large sparse matrix equation system that can be solved by any of a number of well-known sparse matrix solvers
- easy to increase the order of the elements

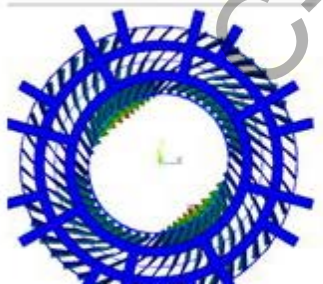


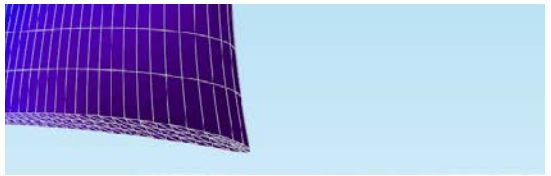
- Structural analysis
- Heat transfer
- Chemical engineering
- Electromagnetics
- Multi-physics
- CFD

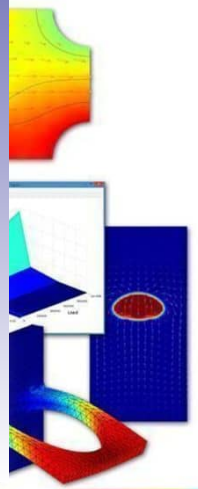


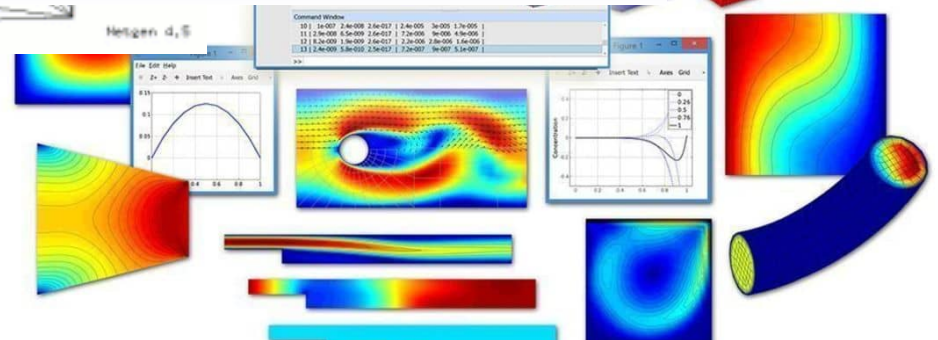






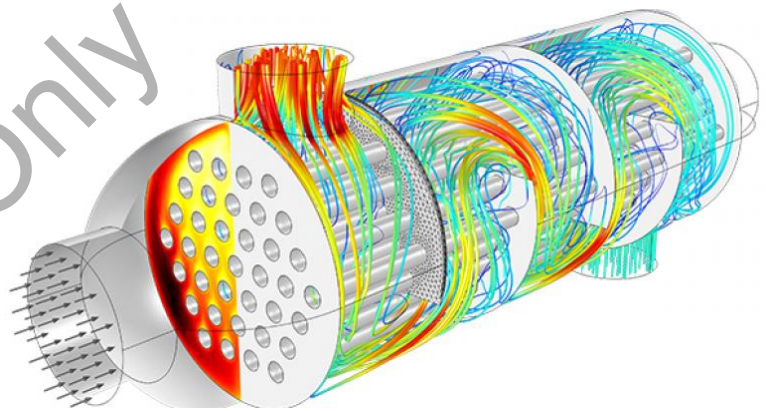






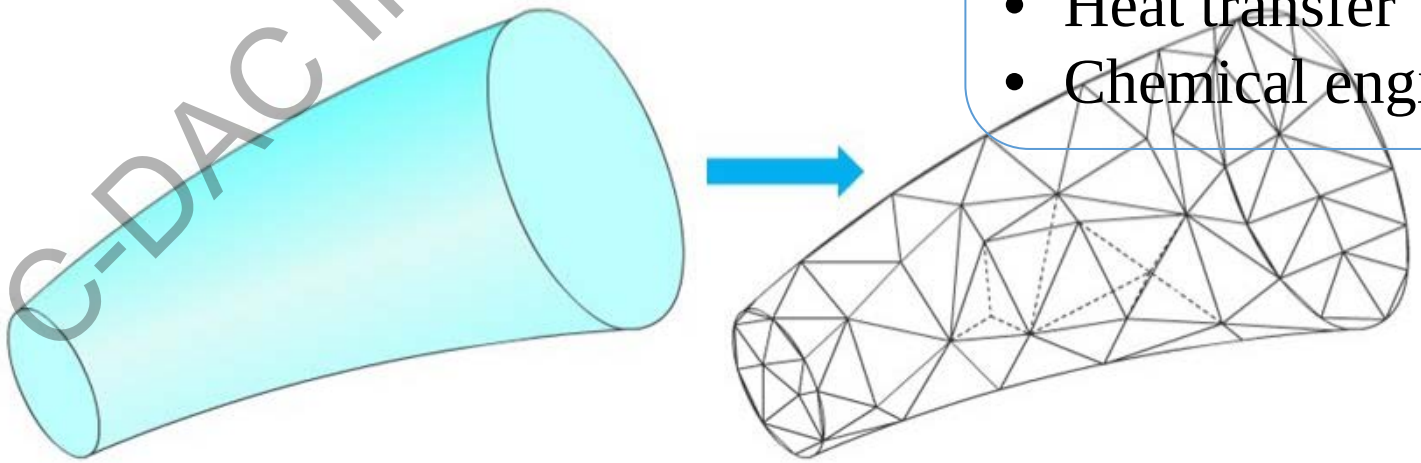
Finite Volume Method

- Subdivision of model is similar to Finite Element method
- Conservation law inherently applied in integral form
- Only needs to do flux evaluation for the cell boundaries
- Has been very successful in solving fluid flow problems
- Local accuracy can be increased by refining the mesh around a corner



Application Areas

- CFD
- Heat transfer
- Chemical engineering



Data reuse and Arithmetic intensity

Processor design is somewhat unbalanced

- loading data is slower than executing the actual operations. This imbalance is large for main memory and less for the various cache levels
- motivated to keep data in cache and keep the amount of *data reuse* as high as possible.

Arithmetic intensity: If ***n*** is the number of data items that an algorithm operates on, and ***f*(*n*)** the number of operations it takes, then the arithmetic intensity is ***f*(*n*)/*n***

Arithmetic intensity is also related to

- ***latency hiding:*** the concept that you can mitigate the negative performance impact of data loading behind computational activity going on
- need more computations than data loads to make this hiding effective
- ***Computational intensity:*** a high ratio of operations per byte/word/number loaded

$$\forall_i:\ x_i \leftarrow x_i + y_i$$

- involves three memory accesses (two loads and one store) and one operation per iteration,
- **Arithmetic intensity = 1/3**

$$\forall_i:\ x_i \leftarrow x_i + a \cdot y_i$$

- two operations, but the same number of memory access since the one-time load of *a* is amortized.
- more efficient than the simple addition, with a **reuse of 2/3**

Inner product calculation

$$\forall_i: s \leftarrow s + x_i \cdot y_i$$

- involves one multiplication and addition per iteration, on two vectors and one scalar.
- only two load operations, since *s* can be kept in register and only written back to memory at the end of the loop. **Reuse here is 1**

Consider the *matrix-matrix product*:

$$\forall_i : c_{ij} = \sum_k a_{ik} b_{kj}$$

- involves $\mathbf{3n^2}$ data items and $\mathbf{2n^3}$ operations, which is of a higher order
- Arithmetic intensity is $\boldsymbol{O(n)}$, meaning that every data item will be used $O(n)$ times
- Implies that, with suitable programming, this operation has the potential of overcoming the bandwidth/clock speed gap by keeping data in fast cache memory.

- matrix-matrix product is the heart of the *LINPACK benchmark*
- An operation that has considerable data reuse
- Relatively insensitive to memory bandwidth and, for parallel computers, properties of the network.
- Typically, computers will attain 60–90% of their *peak performance* on the Linpack benchmark
- Other benchmark may give considerably lower figures

Why should we go parallel?

Two main reasons to use parallelism:

- Reduce the execution time needed to solve a problem
- To be able to solve larger and more complex problems

Other important reasons:

- Computing resources became a commodity and are frequently under-utilized
- Overcome the physical limitations in chip density and production costs of faster sequential computers
- Overcome memory limitations as the solution to some problems require more memory then one could find in just one computer

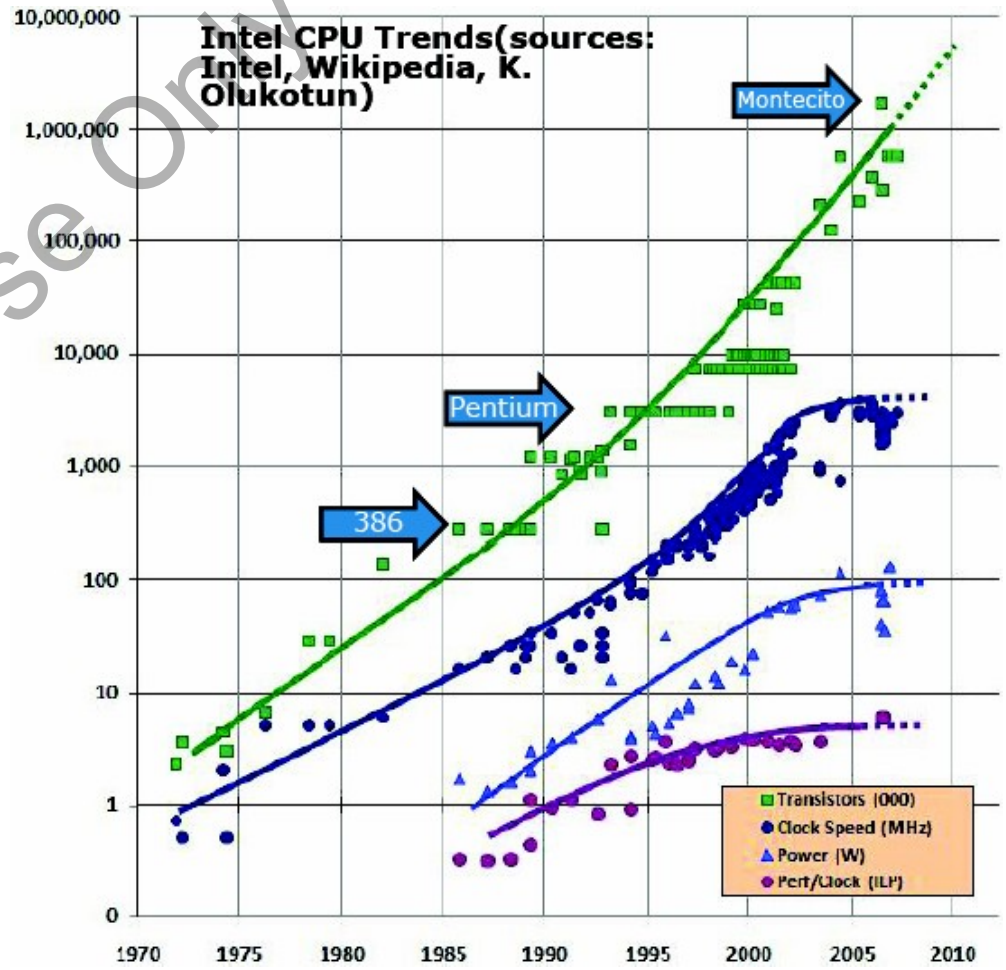
Free Lunch is Over (Herb Sutter, 2005)

Chip density still increasing ~ 2× every 2 years, but

- production is very costly
- clock speeds hit the wall
- heat dissipation / cooling problems

Chips already integrate many parallel ideas

- super-scale execution
- super pipelining
- branch prediction
- out-of-order execution



Solution: Multiple cores on the same chip, i.e. go for parallelism

Implicit Parallelism

Parallelism is exploited implicitly when it is the compiler and runtime system that:

- automatically detects potential parallelism in the program
- assigns the tasks for parallel execution
- controls and synchronizes execution

(+) frees the programmer from the details of parallel execution

(+) it is a more general and flexible solution

(-) very hard to achieve an efficient solution for many applications

Explicit Parallelism

Parallelism is exploited explicitly when it is left to the programmer to:

- annotate the tasks for parallel execution
- assign tasks to processors
- control the execution and the synchronization points

- (+) experienced programmers achieve very efficient solutions for specific problems
- (-) programmers are responsible for all details of execution,
- (-) very hard to achieve an efficient solution for many applications
- (-) programmers must have deep knowledge of the computer architecture to achieve maximum performance.

Parallel Programming

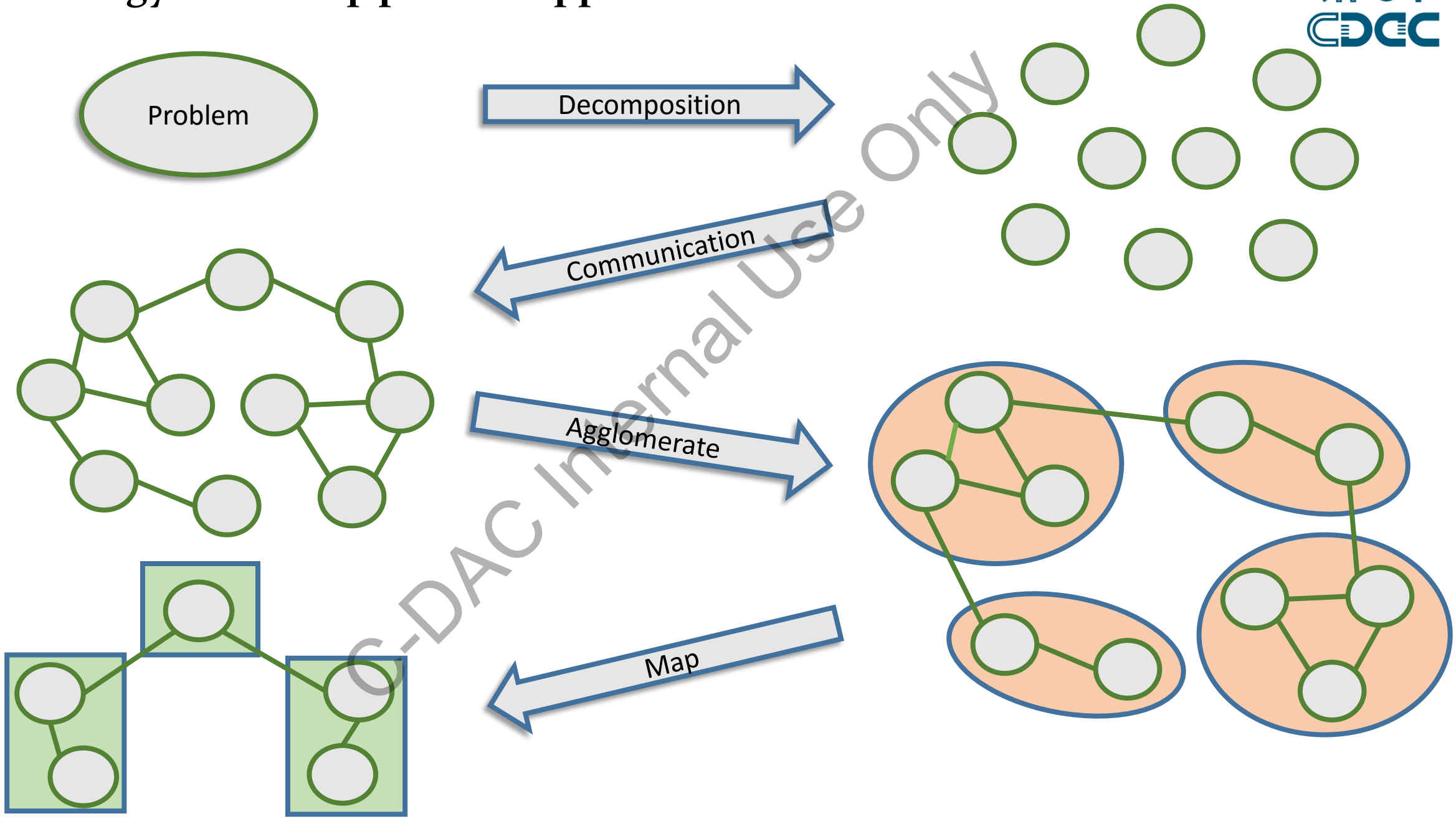
An informal definition:

Parallel programming consists of decomposing a program in smaller parts (or tasks) and efficiently mapping these tasks to available processors for parallel execution.

There are many difficulties:

- How to accomplish decomposition?
- How much should we divide?
- How to map efficiently tasks to processors?
- How to achieve cooperation and/or synchronization of non-independent tasks?
- How to gather results of tasks?
- …

Methodology to develop parallel applications



Communication

The parallel execution of tasks might require:

- **Synchronization** as some tasks may only be executed after some other tasks have completed
- **Communication** between tasks to exchange data (e.g. partial results)

Communication/synchronization can be a limiting factor for performance:

- It has a **cost**. While you communicate, you do not compute!
- **Latency** - minimum time to communicate between two computing nodes
- **Bandwidth** - amount of data we can communicate per unit of time

Avoid communicating too many small messages!

Communication Patterns

Structured communication	• Communication between tasks follows a regular structure (e.g. A tree)
Non-structured communication	• Communication between tasks follows an arbitrary graph
Static communication	• Communication pattern between tasks is known before execution
Dynamic communication	• Communication between tasks is only determined during execution

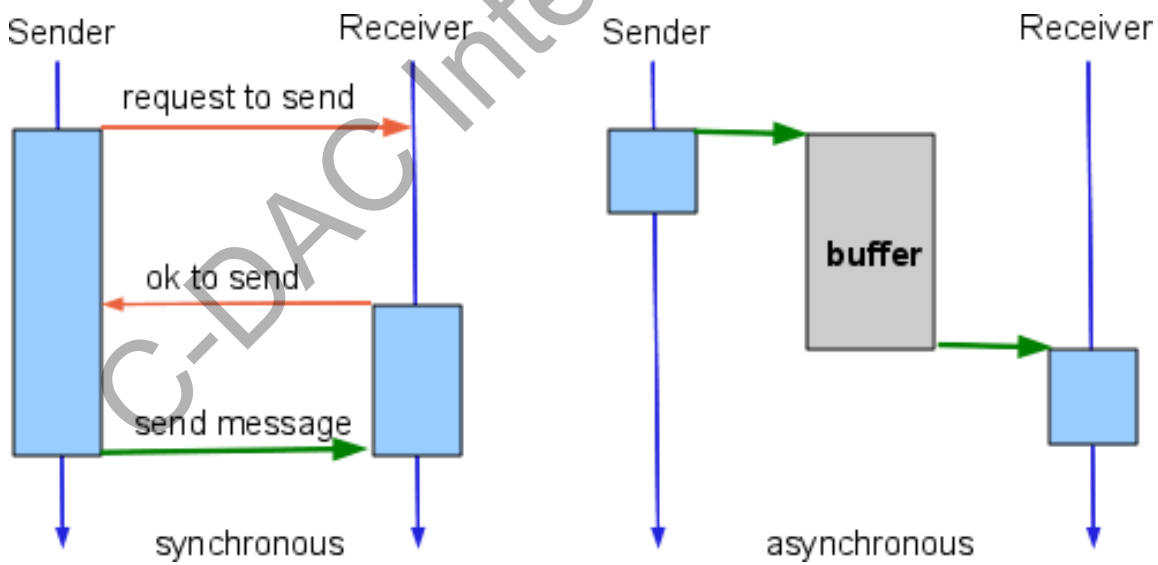
Communication Patterns

Synchronous communication

- sender and receiver have to synchronize to
- start communicating

Asynchronous communication

- sender writes messages to a buffer and continues execution; when ready, the receiver reads the messages from the buffer.
- no agreement needed



Aggregation

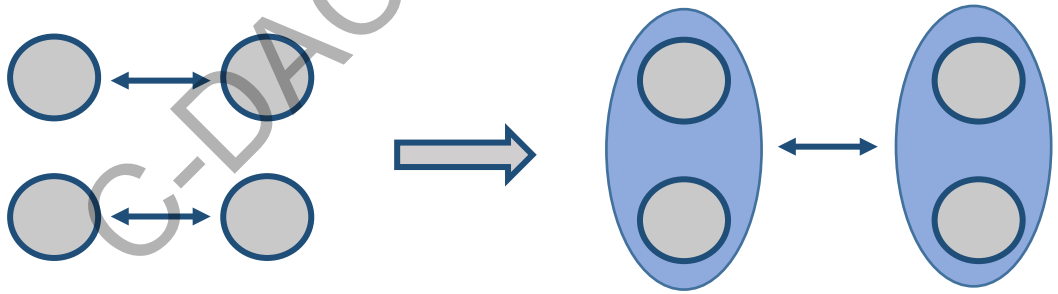
How small can the tasks be for parallel execution?

- if too small, we incur in higher communication costs
- time to compute the task must be higher then time to communicate it

Granularity of tasks- measures the ratio between the average time to compute and to communicate the tasks

- avoid too fine grained tasks!

Aggregating small tasks into larger ones helps to reduce communication costs.



but, also do not over do it as with too large tasks we might limit available parallelism.

Granularity

Granularity measures the size of a task.

- it can be fine grain, medium grain, or coarse grain
- The main question is…. **Which task size maximizes performance?**

Fine granularity:

- smaller task sizes, less computation and more communication, thus smaller ratio between computation and communication.
- **(+)** simplifies efficient workload balancing.
- **(-)** computation cost of one task may not compensate for parallel costs (task creation, communication and synchronization).

Coarse granularity:

- larger ratio between computation and communication, thus, in principle, better efficiency.
- **(-)** makes it difficult to achieve efficient workload balancing.
- **(+)** computation costs of tasks compensate for parallel costs.

Mapping/Scheduling

To achieve maximum performance, one should:

- maximize processor utilization (keep them busy computing the tasks), and
- minimize communication/synchronization between processors.

Thus, the question is how to best assign tasks to available processors to achieve maximum performance?

- ***static scheduling***- can be predetermined at compile time, normally with regular data-parallelism
- ***dynamic scheduling***- take decisions during execution and try to load balance work among the processors

Scheduling – cost of tasks

Cost of tasks (or granularity) influences decisions:

Easy: The task all have equal (unit) cost

Branch free loops

Harder: The tasks have different, but known times

Sparse matrix – vector multiply

Hardest: The task costs unknown until after execution

circuits, search

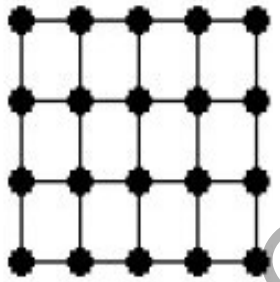
Scheduling - dependencies between tasks

Dependency between tasks influences decisions:

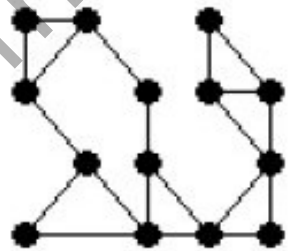
Easy: The tasks once created, don not, communicate

Embarrassingly Parallel

Harder: The tasks communicate in a predictable pattern



Regular



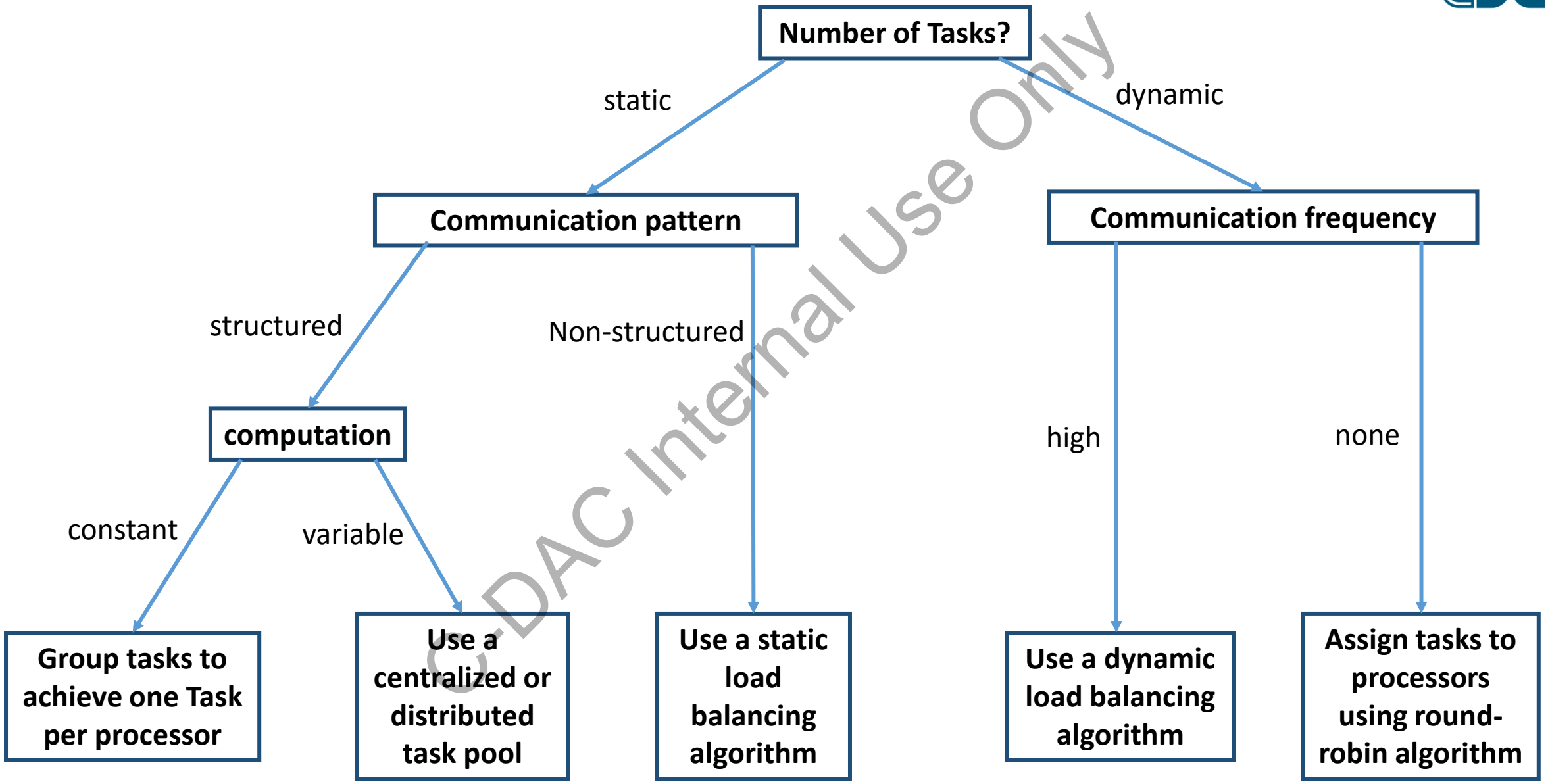
Irregular

PDE Solver

Hardest: The communication pattern is unpredictable

Discrete Event Simulation

Load Balancing / Workload Balancing



Main Parallel Programming Paradigms

The following paradigms are the most commonly used:

- Master/Worker
- Single Program Multiple Data (SPMD)
- Data Pipelining
- Divide-and-Conquer or Tree-search

Which paradigm should we use?

- Depends on the problem
- Type of parallelism: data or functional parallelism
- Type of resources available, which might influence the granularity that can be exploited.

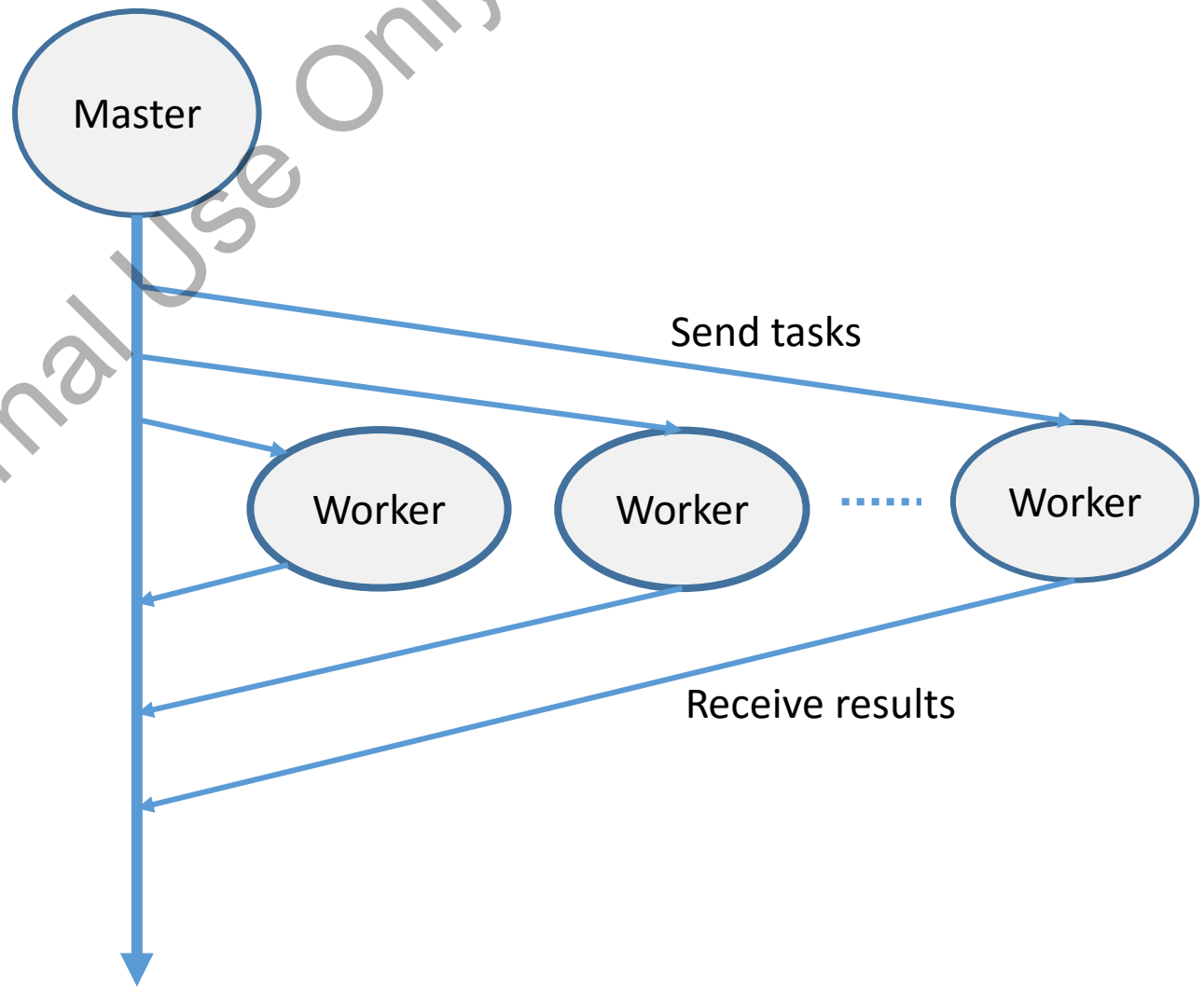
Master/Worker

Master process is responsible for

- initiating the computation
- possibly determine the tasks
- distribute tasks to worker processors
- aggregate partial results from workers, and produce final result

Workers execute a simpler execution cycle

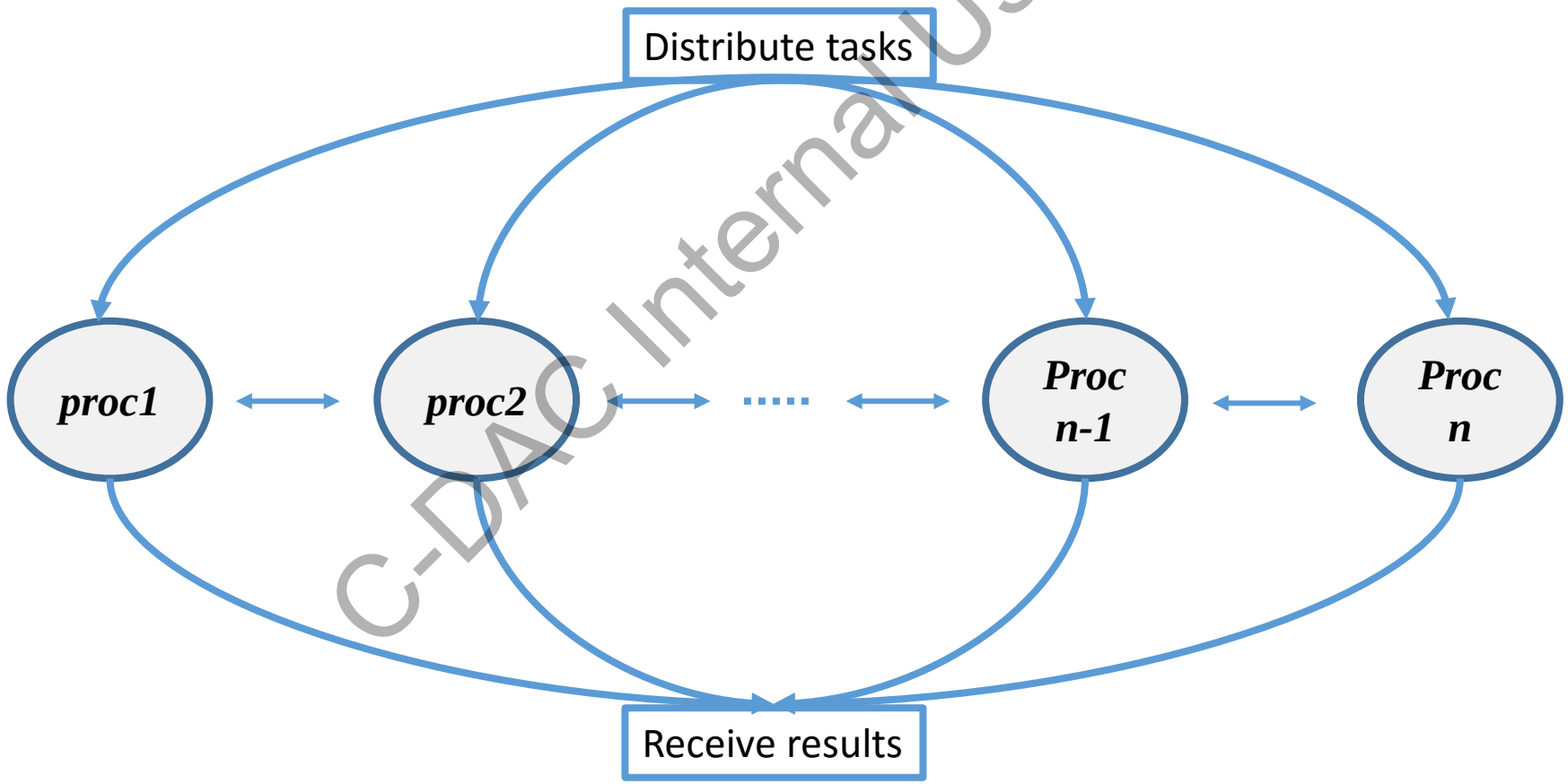
- receive task
- compute task
- send task-result to master



Single Program Multiple Data

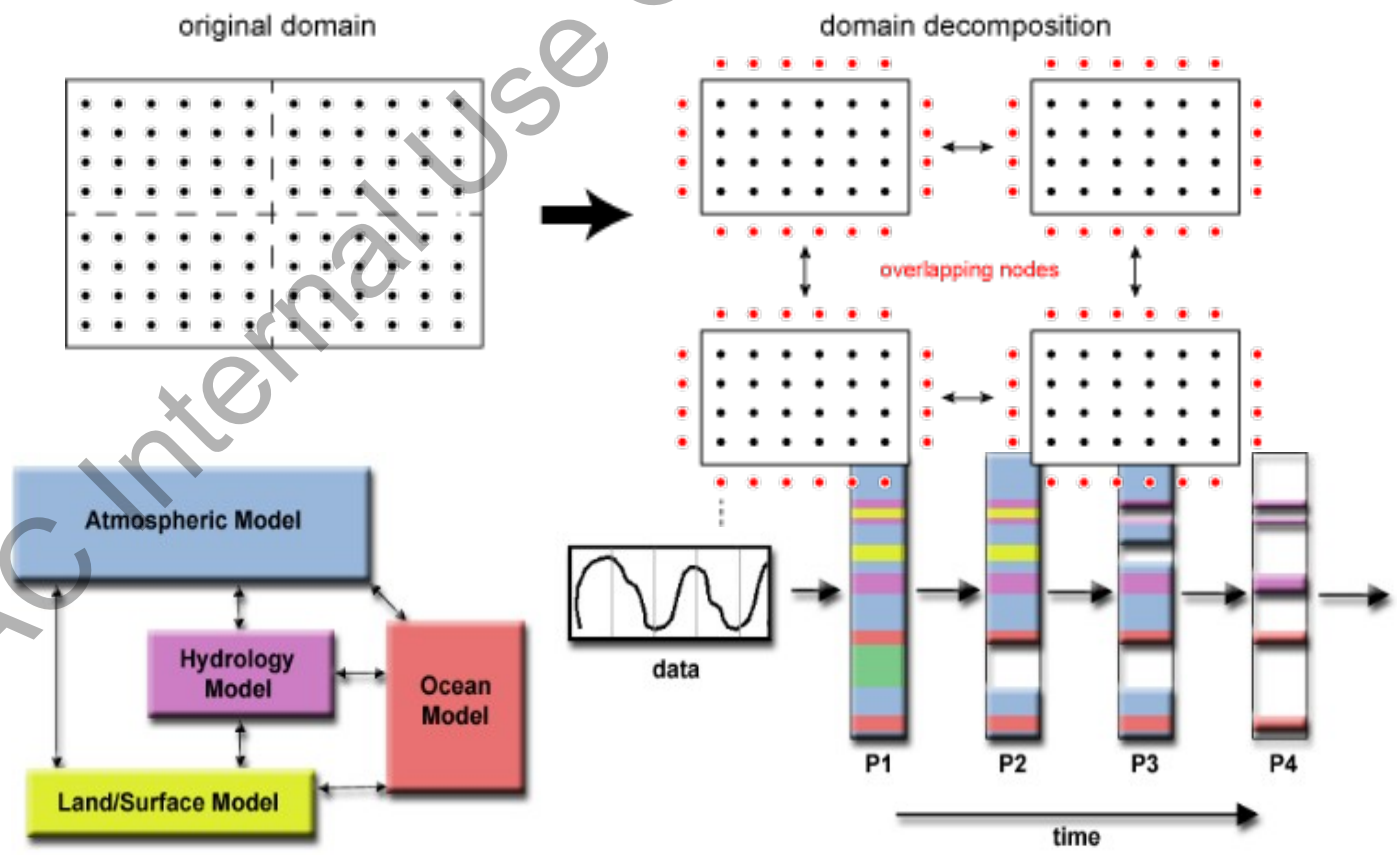
All processes execute the same program, but on different parts of data.

- also known as data parallelism
- similar to Master/Worker, but here we might have communication between tasks.



Domain Decomposition

Functional Decomposition



Climate Modelling

Data Analysis

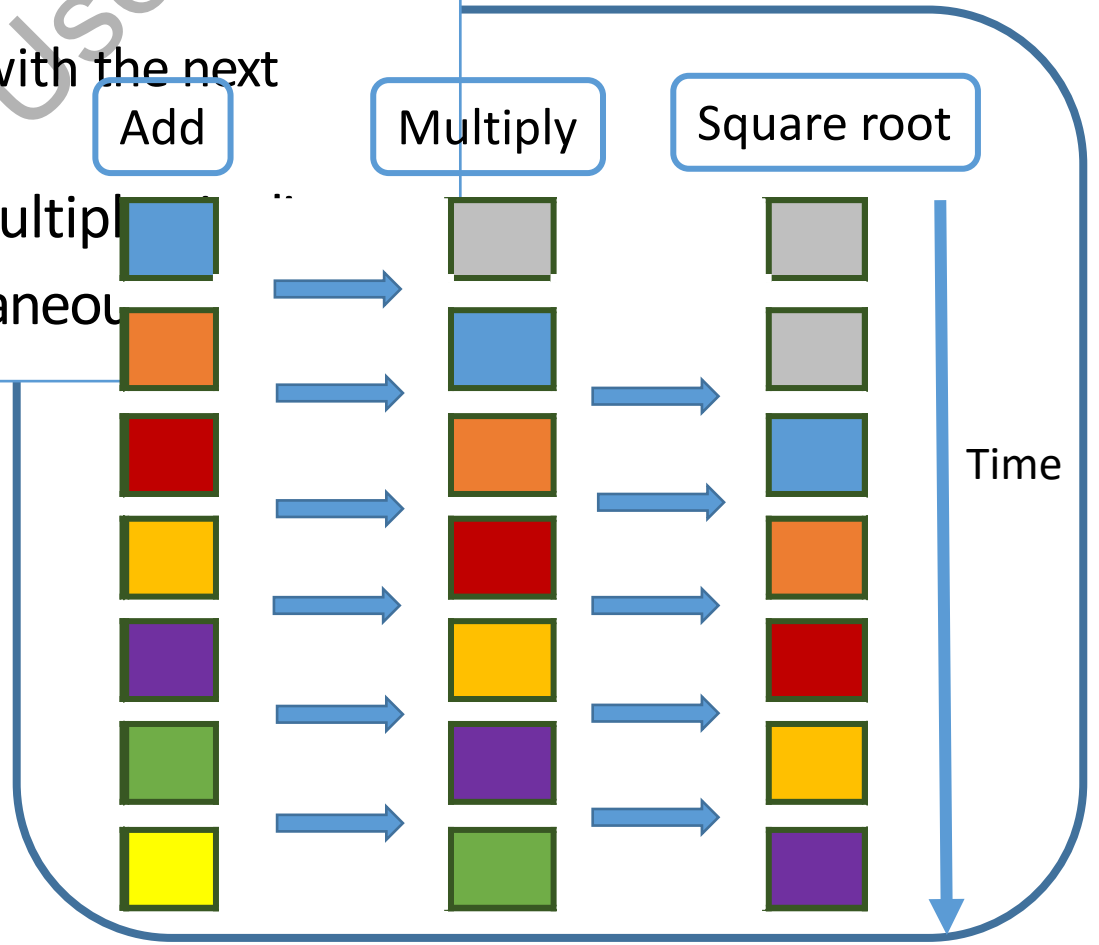
Data Pipelining

Processes are organized in a pipeline fashion

- for each task, each process does part of the computation
- each process only communicates with the next process in the pipeline

Parallelism is achieved by having multipl

(i.e. tasks) being executed simultaneou



Example:

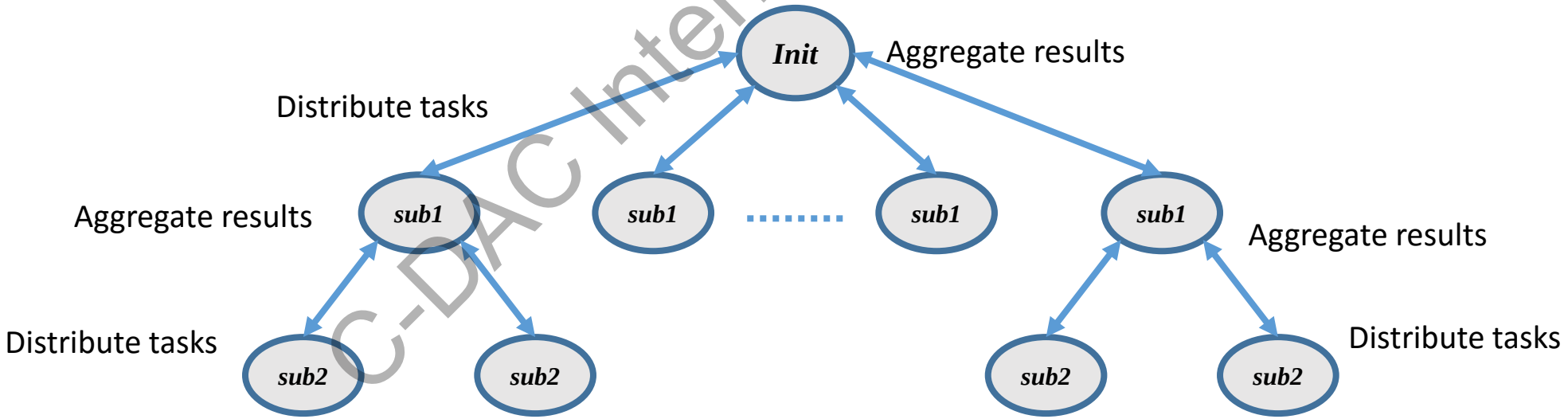
$$F(x,y) = \sqrt{93(x+y)}$$

Input values : (x,y)

Divide & Conquer Parallelism

- Solution space corresponds to a search tree
- Divides a problem in similar but less complex sub-problems
- Solve sub-problems one easily solves initial problem

- Normally computationally intensive
- Allow for a variety of parallelization strategies
- Require dynamic load balancing



Thank You

Speculative Parallelism

Used when dependencies are too complex that other pradigms are not sufficient.

Parallelism is introduced by performing speculative computations

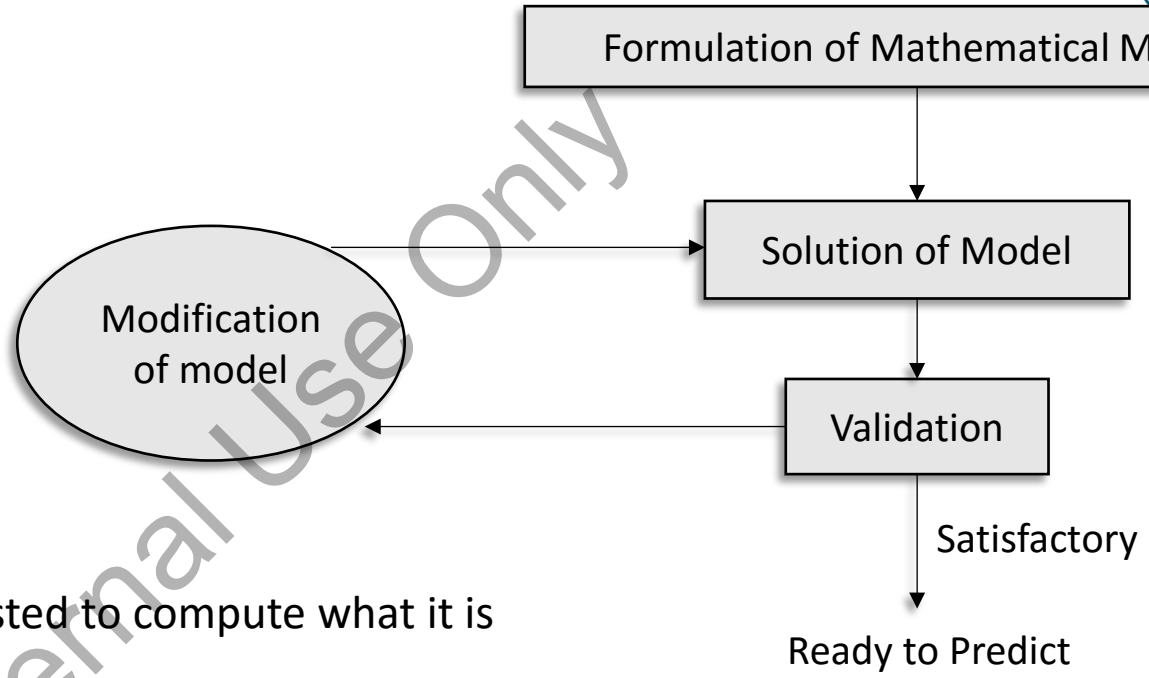
- some related computations are anticipated, taking an optimistic assumption that they will be necessary.
- if they are not necessary, they are terminated and some prior computation state may have to be recovered.
- common in association with branch-and-bound algorithms

Programming Paradigms

The programming paradigms can also be differentiated by employing static or dynamic strategies for decomposition and/or mapping:

	Decomposition	Mapping
Master/Slave	static	dynamic
Single Program Multiple Data (SPMD)	static	static/dynamic
Data Pipelining	static	static
Divide and Conquer	dynamic	dynamic
Speculative Parallelism	dynamic	dynamic

Good Programs



1. *Reliability:* code does not have errors and can be trusted to compute what it is supposed to compute.
2. *Robustness:* The code has a wide range of applicability
3. *Portability :* The code can be transferred from one computer to another with a minimum effort and without losing reliability
4. *Maintainability : Any change* should be possible with minimum effort

- Is often used in place of experiments when experiments are too large, too expensive, too dangerous, or too time consuming.
- Can be useful in “what if” studies; e.g. to investigate the use of pathogens (viruses, bacteria, fungi) to control an insect population.
- Is a modern tool for scientific investigation.

Numerical Simulation Process

- Geometry Modelling
- Mathematical modelling
- Discretization Method
- Grid Generation
- Numerical Solution
- Post Processing
- Validation

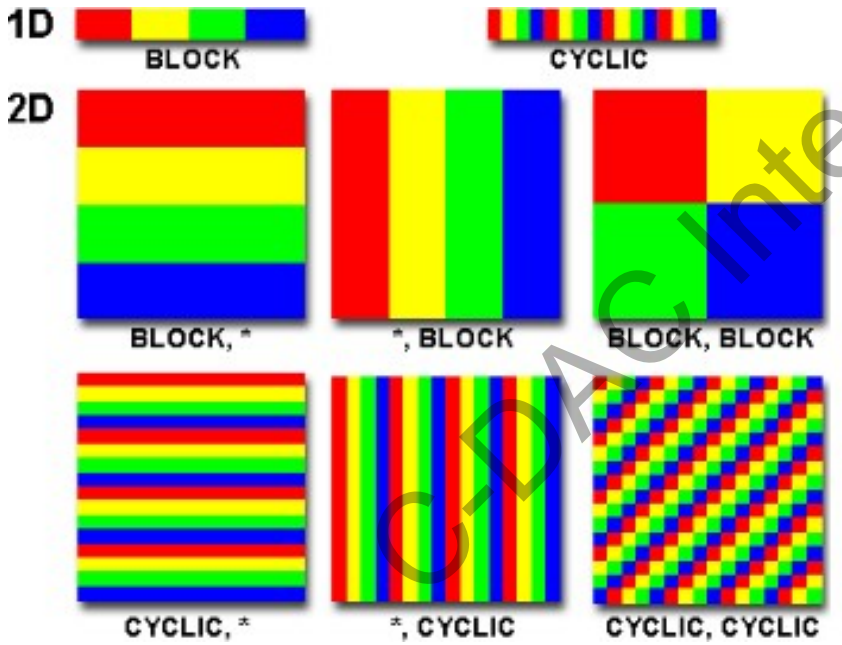
Decomposition

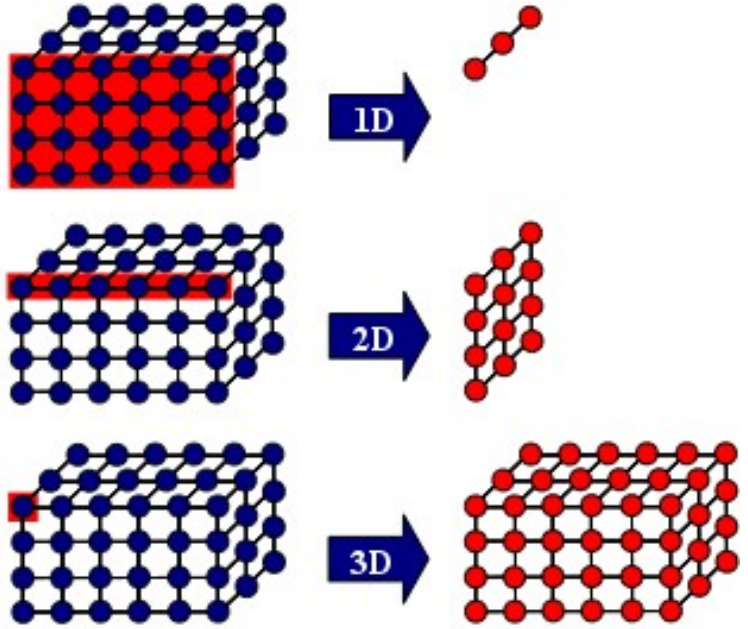
- Decomposing a problem into smaller problems, not only helps in reducing the complexity of the problem, but also allows for the sub-problems to be executed in parallel.
- There are two main strategies to decompose a problem:
  - Domain decomposition: decomposition based on the data.
  - Functional decomposition: decomposition based on the computation.
- A good scheme divides either data or computation, or both, into smaller tasks.

Domain Decomposition

First the data is partioned and only after we associate the computation to partitions.

All tasks execute the same operations on different parts of data (data-parallelism).

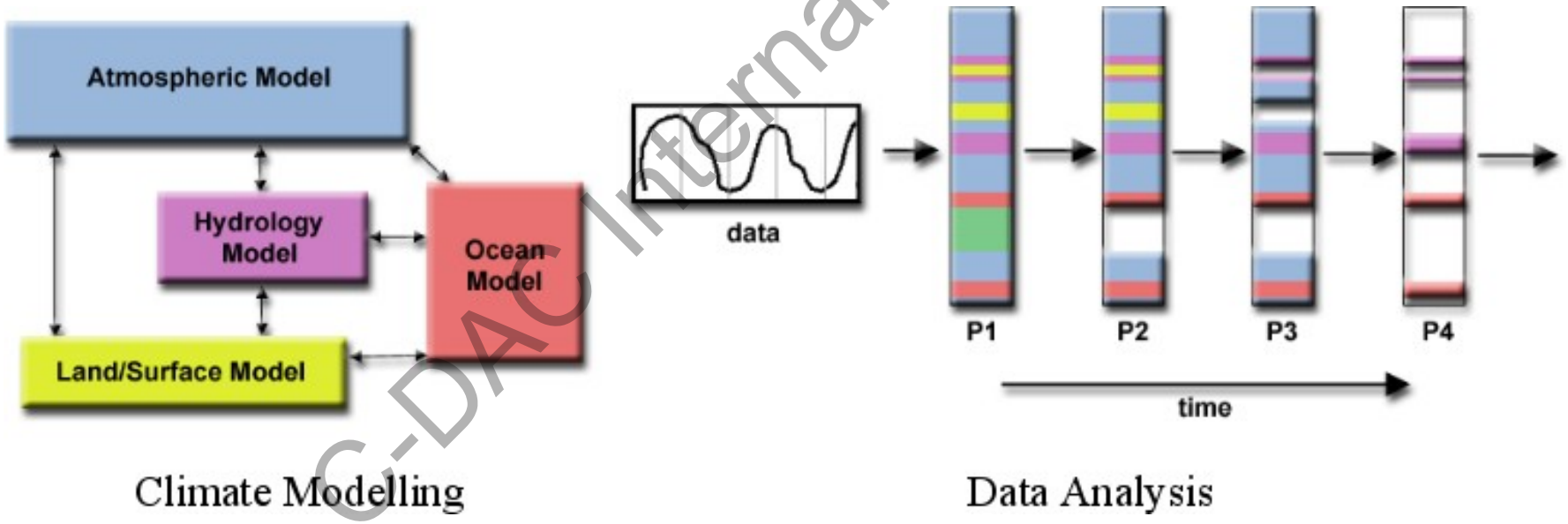




Functional Decomposition

First we divide the computation in tasks and only after associate data with tasks.

Different tasks may execute different operations on different data (functional parallelism).



Climate Modelling

Data Analysis

Communication

The parallel execution of tasks might require:

- synchronization as some tasks may only be executed after some other tasks have completed
- communication between tasks to exchange data (e.g. partial results)

Communication/synchronization can be a limiting factor for performance:

- it has a cost. While you communicate, you do not compute!
- latency - minimum time to communicate between two computing nodes
- bandwidth - amount of data we can communicate per unit of time

Avoid communicating too many small messages!

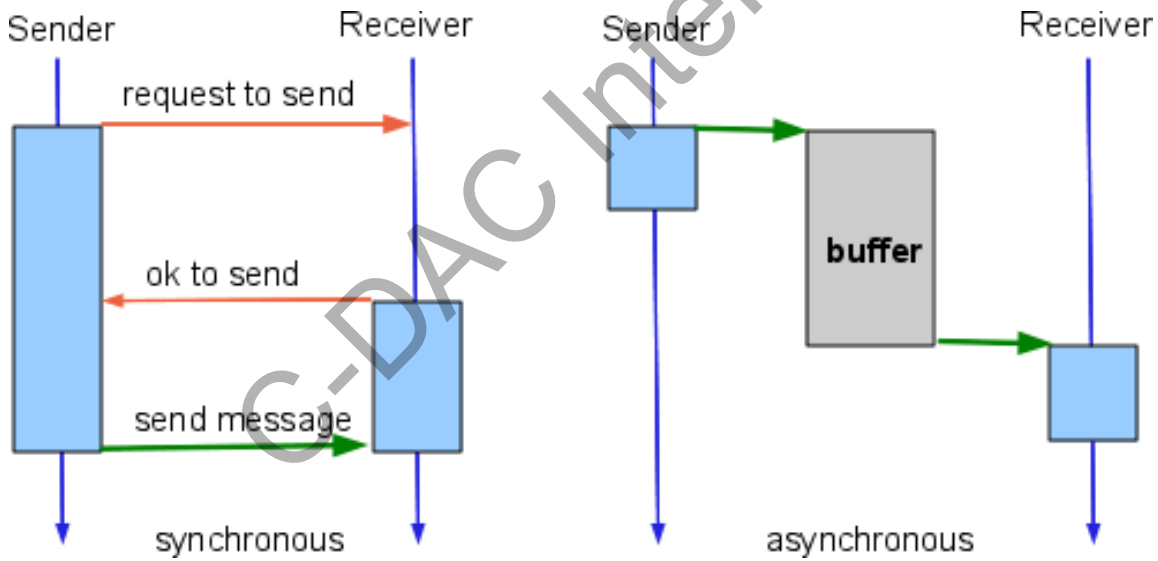
Communication Patterns

Synchronous communication

- sender and receiver have to synchronize to start communicating (rendez-vous protocol)

Asynchronous communication

- sender writes messages to a buffer and continues execution; when ready, the receiver reads the messages from the buffer.
- no agreement needed



Aggregation

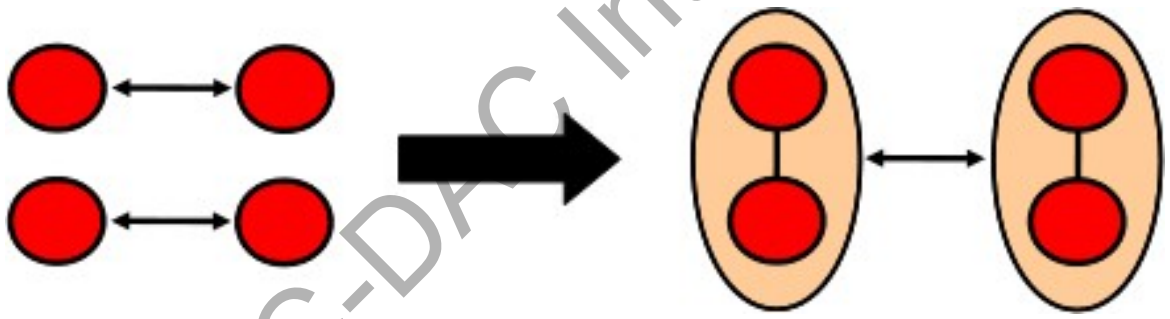
How small can the tasks be for parallel execution?

- if too small, we incur in higher communication costs
- time to compute the task must be higher then time to communicate it

Granularity of tasks- measures the ratio between the average time to compute and to communicate the tasks

- avoid too fine grained tasks!

Aggregating small tasks into larger ones helps to reduce communication costs.



but, also do not over do it as with too large tasks we might limit available parallelism.

Granularity

Granularity measures the size of a task.

- it can be fine grain, medium grain, or coarse grain
- The main question is? Which task size maximizes performance?

Fine granularity:

- smaller task sizes, less computation and more communication, thus smaller ratio between computation and communication.
- (+) simplifies efficient workload balancing.
- (-) computation cost of one task may not compensate for parallel costs (task creation, communication and synchronization).

Coarse granularity:

- larger ratio between computation and communication, thus, in principle, better efficiency.
- (-) makes it difficult to achieve efficient workload balancing.
- (+) computation costs of tasks compensate for parallel costs.

Mapping/Scheduling

To achieve maximum performance, one should:

- maximize processor utilization (keep them busy computing the tasks), and
- minimize communication/synchronization between processors.

Thus, the question is how to best assign tasks to available processors to achieve maximum performance?

- static scheduling- can be predetermined at compile time, normally with regular data-parallelism
- dynamic scheduling- take decisions during execution and try to load balance work among the processors

Scheduling – cost of tasks

Cost of tasks (or granularity) influences decisions:

Easy: The task all have equal (unit) cost

n items

p bins

Branch free loops

Harder: The tasks have different, but known times

Sparse matrix – vector multiply

n items

p bins

Hardest: The task costs unknown until after execution

GCM, circuits, search

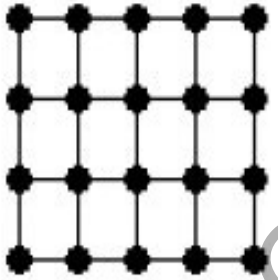
Scheduling - dependencies between tasks

Dependency between tasks influences decisions:

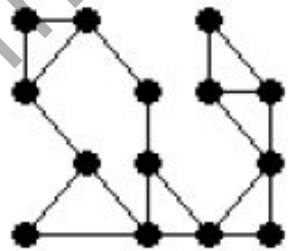
Easy: The tasks once created, don not, communicate

Embarrassingly Parallel

Harder: The tasks communicate in a predictable pattern



Regular



Irregular

PDE Solver

Hardest: The communication pattern is unpredictable

Discrete Event Simulation