

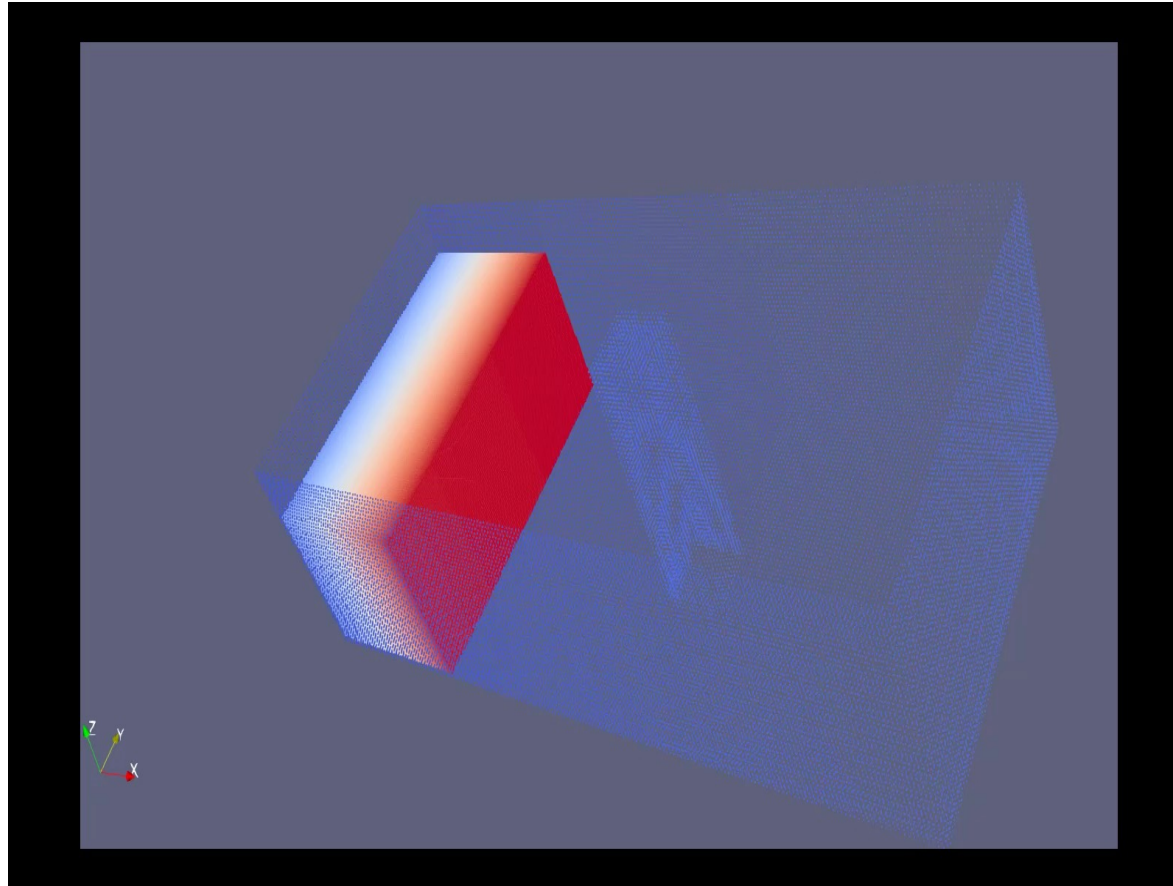


Good Practices For (Parallel) Programing

Carlos Jaime Barrios Hernández, PhD

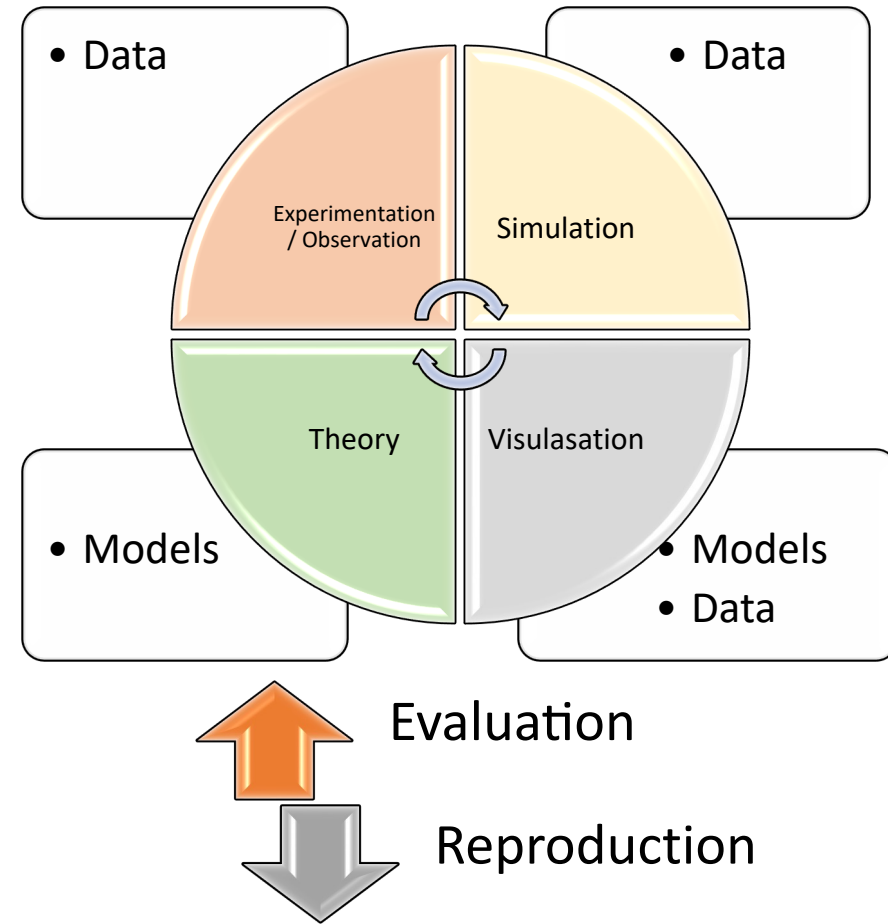
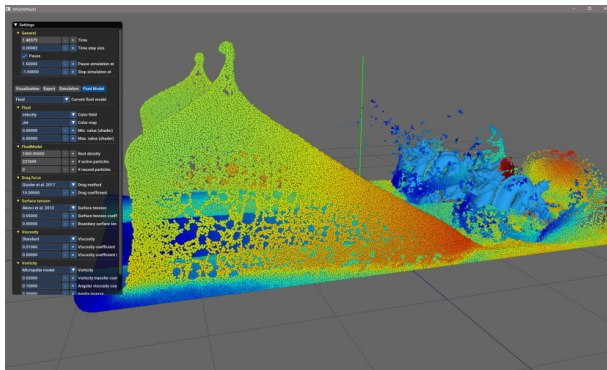
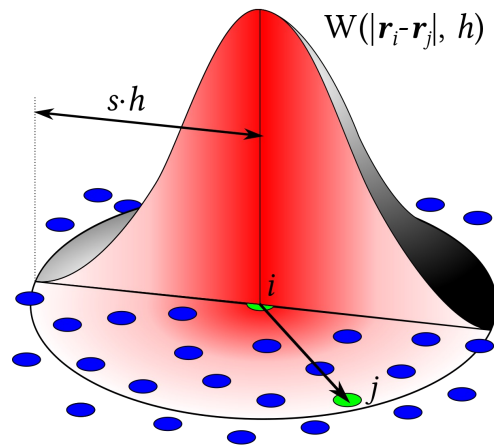
@carlosjaimebh

Realism and Performance



<https://dual.sphysics.org/>

The Challenge



Visualisation

- Accuracy
- Utility

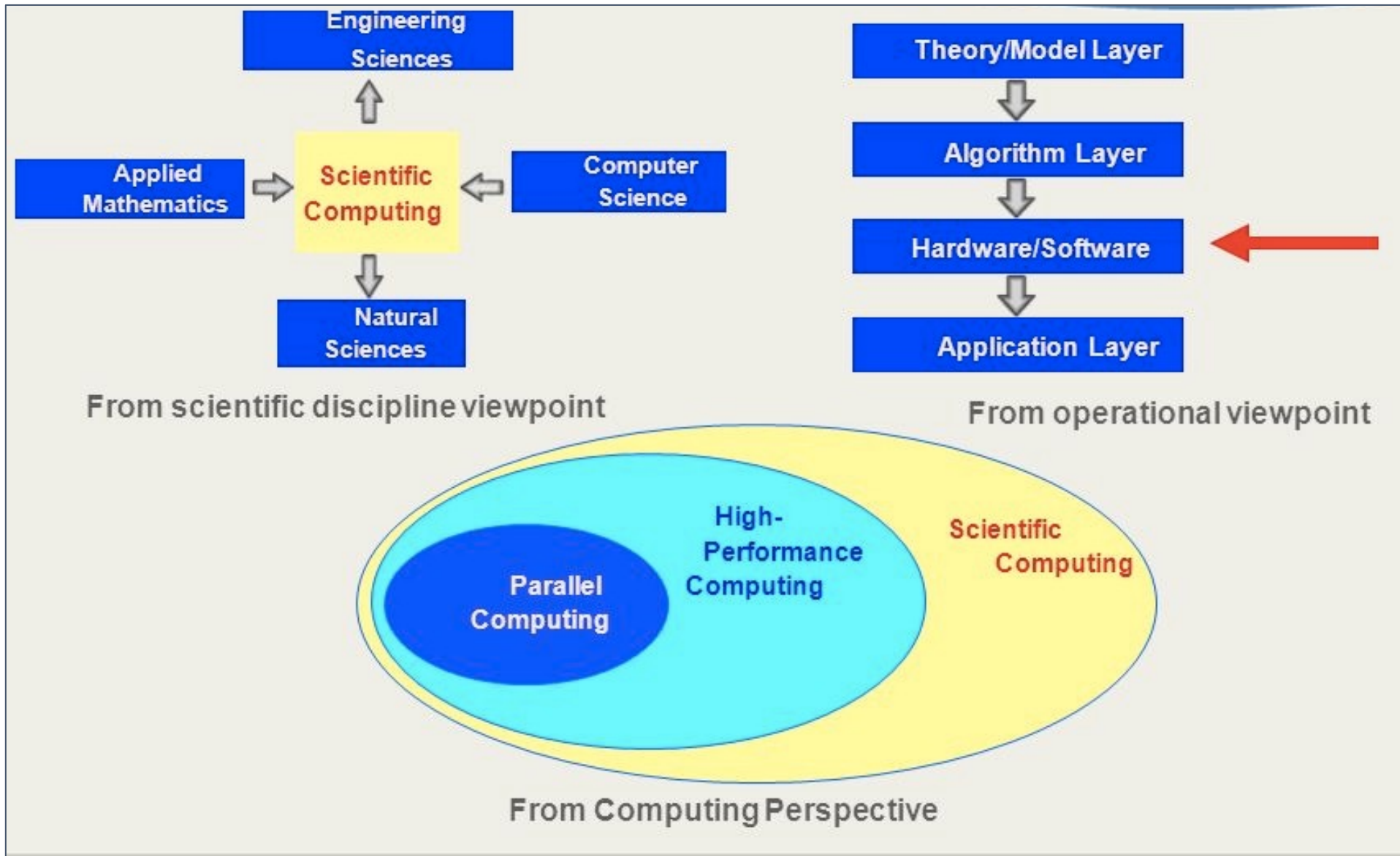
Numerical Analysis

- Reproducibility
- Reliability

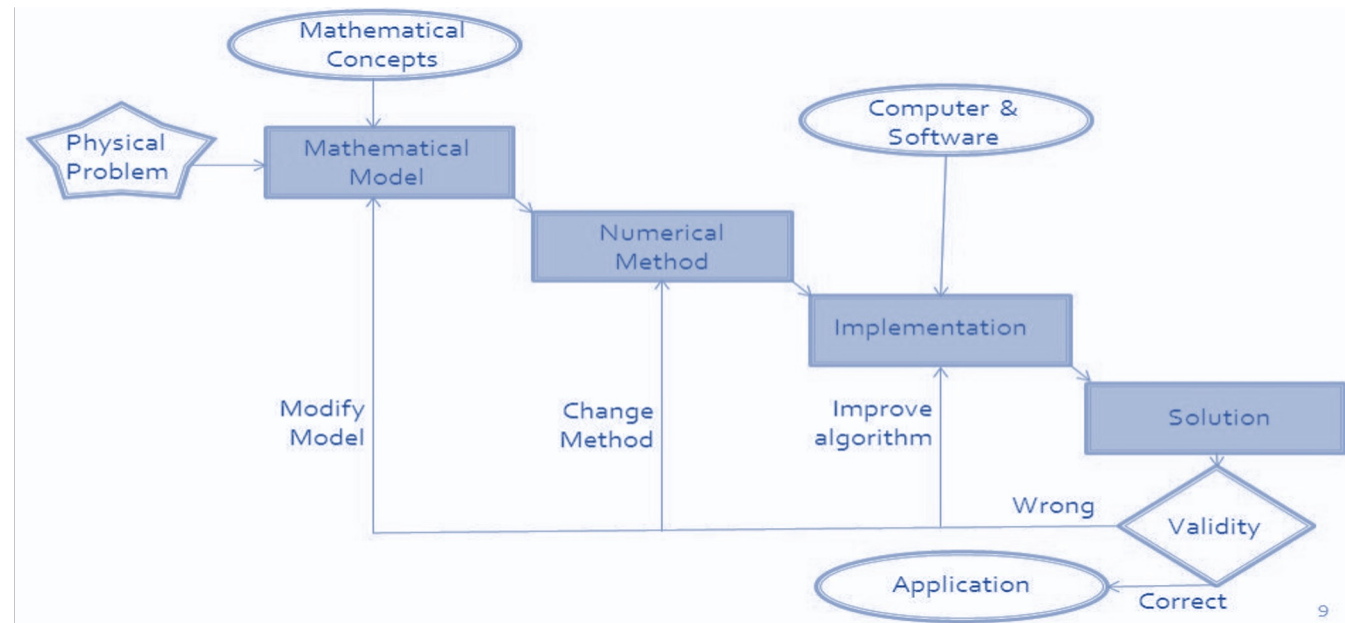
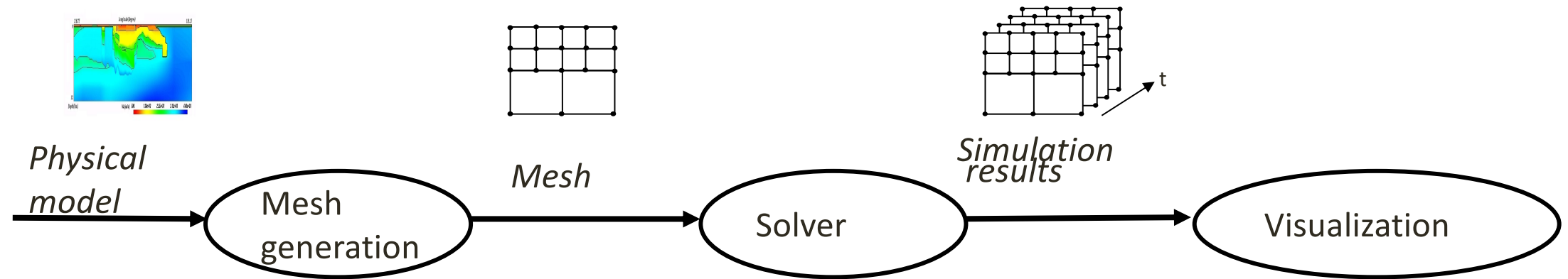
Implementation

- Efficiency
- Performance

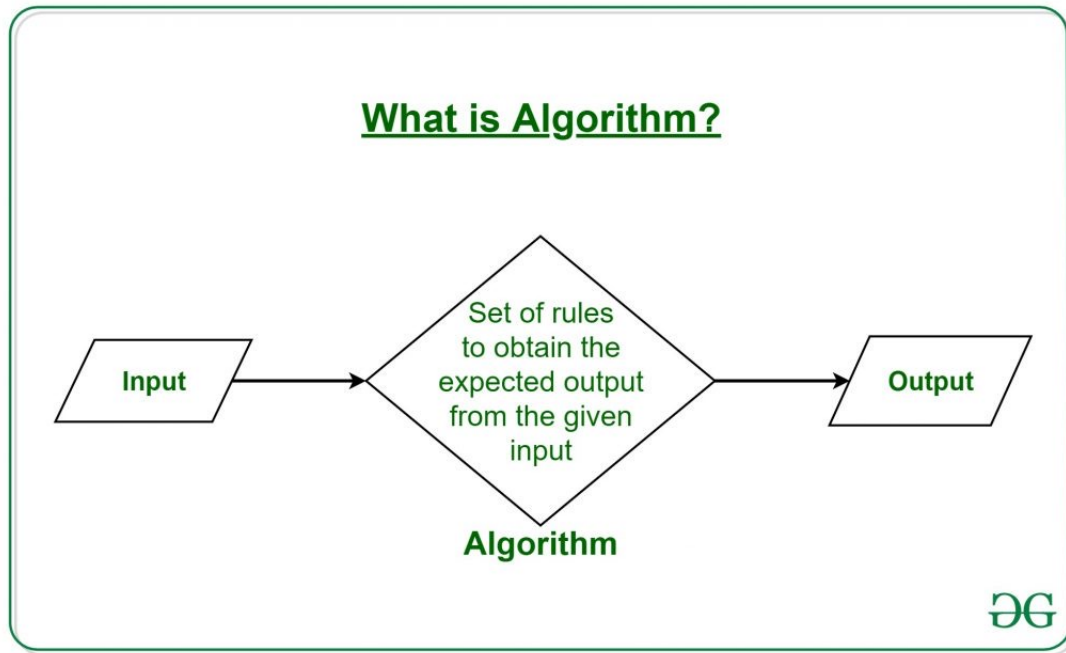
What is Scientific Computing?



The Scientific Computing Process

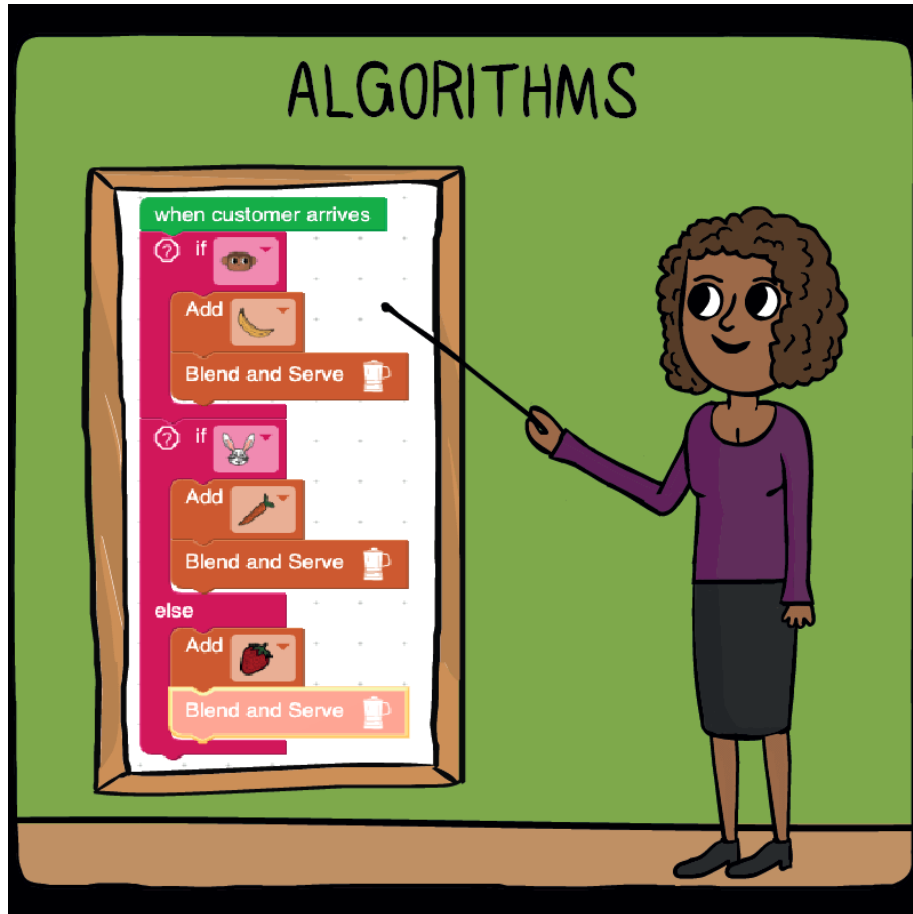


Introducing Algorithms...



- The word Algorithm means “a process or set of rules to be followed in calculations or other problem-solving operations”. Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

Introducing Algorithms...



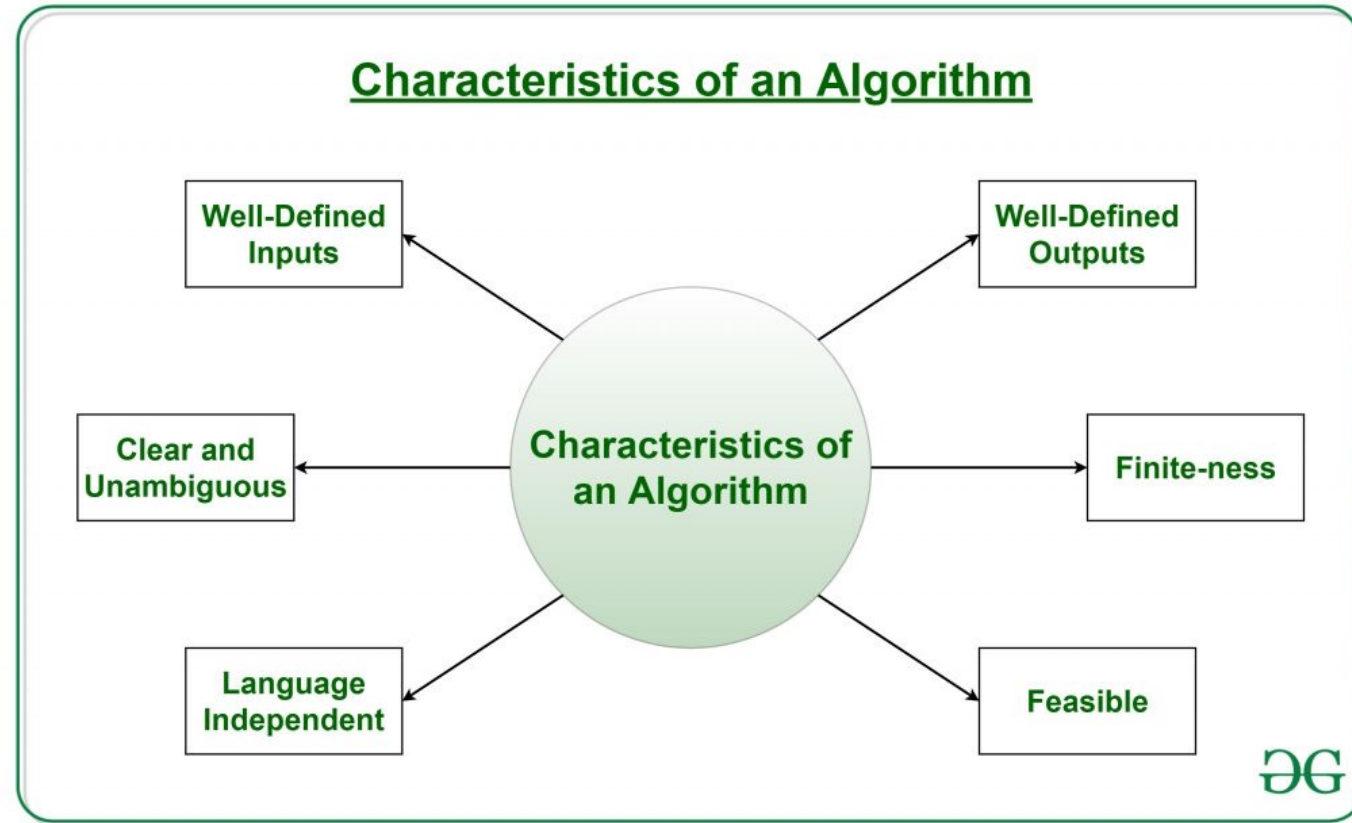
- Similarly, algorithms help to do a task in programming to get the expected output. The Algorithms designed are language or implementation independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.
 - Language implementation (or other implementation) is a translation to be executed in a context (or runtime).

From: <https://www.geeksforgeeks.org/introduction-to-algorithms/>

<https://en.wikipedia.org/wiki/Algorithm>

Characteristics of an Algorithm

- **Clear and Unambiguous:** Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs:** If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.



From: <https://www.geeksforgeeks.org/introduction-to-algorithms/>

<https://en.wikipedia.org/wiki/Algorithm>

Possible Analysis of an Algorithm

- **Priori Analysis:** “Priori” means “before”. Hence Priori analysis means checking the algorithm before its implementation. In this, the algorithm is checked when it is written in the form of theoretical steps. This Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation. This is done usually by the algorithm designer. It is in this method, that the Algorithm Complexity is determined.
- **Posterior Analysis:** “Posterior” means “after”. Hence Posterior analysis means checking the algorithm after its implementation. In this, the algorithm is checked by implementing it in any programming language and executing it. This analysis helps to get the actual and real analysis report about correctness, space required, time consumed etc.
 - **Time Factor:** Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
 - **Space Factor:** Space is measured by counting the maximum memory space required by the algorithm.

Complexity (1/2)

- **Space Complexity:** Space complexity of an algorithm refers to the amount of memory that this algorithm requires to execute and get the result. This can be for inputs, temporary operations, or outputs.

How to calculate Space Complexity?

The space complexity of an algorithm is calculated by determining following 2 components:

- **Fixed Part:** This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size, etc.
- **Variable Part:** This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.

Complexity (2/2)

- **Time Complexity:** Time complexity of an algorithm refers to the amount of time that this algorithm requires to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

How to calculate Time Complexity?

The time complexity of an algorithm is also calculated by determining following 2 components:

- **Constant time part:** Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, etc.
- **Variable Time Part:** Any instruction that is executed more than once, say n times, comes in this part. For example, loops, recursion, etc.

From Space Complexity to Big O Notation

- The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely.
- Similar to time complexity, space complexity is often expressed asymptotically in **big O** notation, such $O(n)$, $O(n \log n)$, $O(n^\alpha)$, $O(2^n)$, etc., where n is a characteristic of the input influencing space complexity.
- Analogously to time complexity classes DTIME(f(n)) and NTIME(f(n)), the complexity classes DSPACE(f(n)) and NSPACE(f(n)) are the sets of languages that are decidable by deterministic (respectively, non-deterministic) Turing machines that use $O(f(n))$ space.
- The complexity classes PSPACE and NPSPACE allow f to be any polynomial, analogously to P and NP. That is,

$$\text{PSPACE} = \bigcup_{c \in \mathbb{Z}^+} \text{DSPACE}(n^c)$$

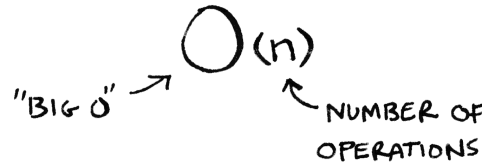
and

$$\text{NPSPACE} = \bigcup_{c \in \mathbb{Z}^+} \text{NSPACE}(n^c)$$

From: https://en.wikipedia.org/wiki/Space_complexity

A little about the Big O Notation

- Big O notation is special notation that tells you how fast an algorithm is.



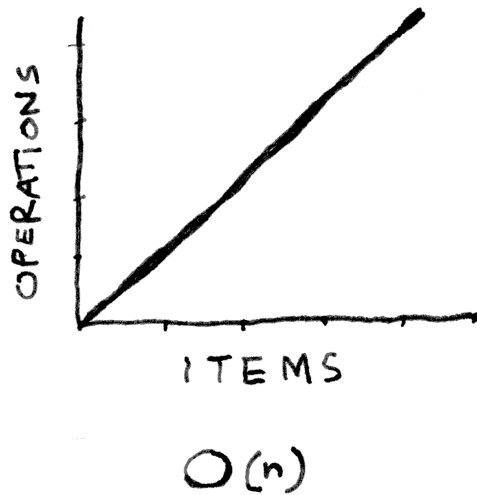
- Big O doesn't tell you the speed in seconds. Big O notation lets you compare the number of operations. It tells you how fast the algorithm grows.
- This tell you the number of operations an algorithm will make. It's called Big O notation because you put a « big O » in front of the number of operations.
- **Also, Big O notation is called as Bachmann–Landau notation or asymptotic notation.**

From: https://github.com/egonSchiele/grokking_algorithms
https://en.wikipedia.org/wiki/Big_O_notation

Running Time

- It's the running phase of an algorithm

Linear Time



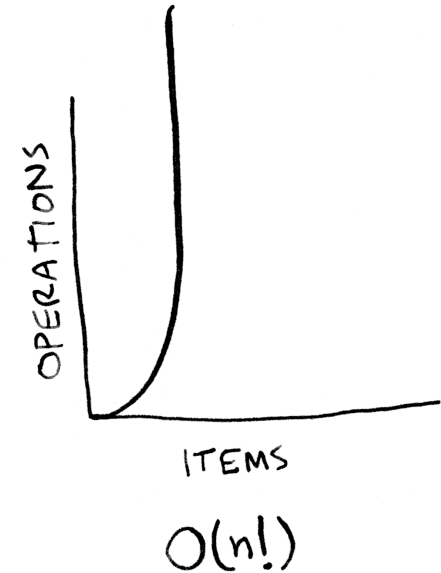
An algorithm is said to take **linear time**, or $O(n)$ time, if its time complexity is $O(n)$. Informally, this means that the running time increases at most linearly with the size of the input.

Logarithmic Time



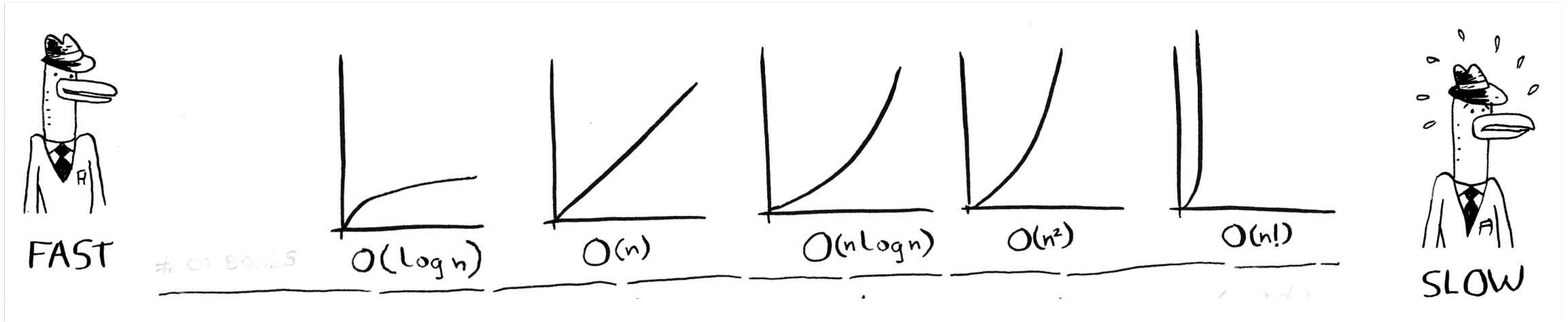
An algorithm is said to take **logarithmic time** when $T(n) = O(\log n)$. Algorithms taking logarithmic time are commonly found in operations on [binary trees](#) or when using [binary search](#).

Factorial Time

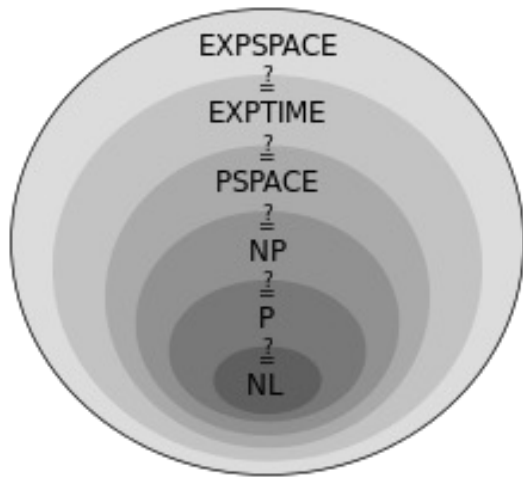


Recall that a **factorial** is the **product of the sequence of n integers**. For example, the factorial of 5, or $5!$, is: $5 * 4 * 3 * 2 * 1 = 120$. We will find ourselves writing algorithms with factorial time complexity when calculating permutations and combinations.

From Fast to Slow Algorithms...



- **$O(\log n)$** also known as log time . Example: Binary Search
- **$O(n)$** also known as linear time. Example: Simple Search
- **$O(n \log n)$** . Example: a fast sorting algorithm like Quicksort
- **$O(n^2)$** . Example: a slow sorting algorithm, like Selection Sort
- **$O(n!)$** . Example a really slow algorithm , like the traveling salesperson.

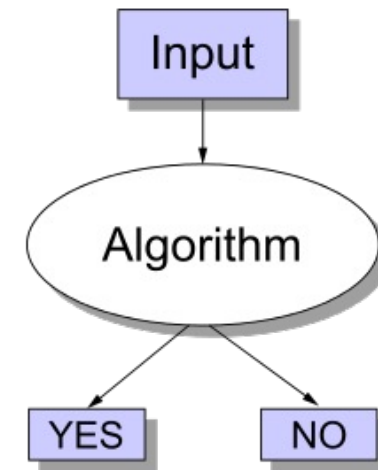


A representation of the relationships between several important complexity classes

And a Little more of Complexity

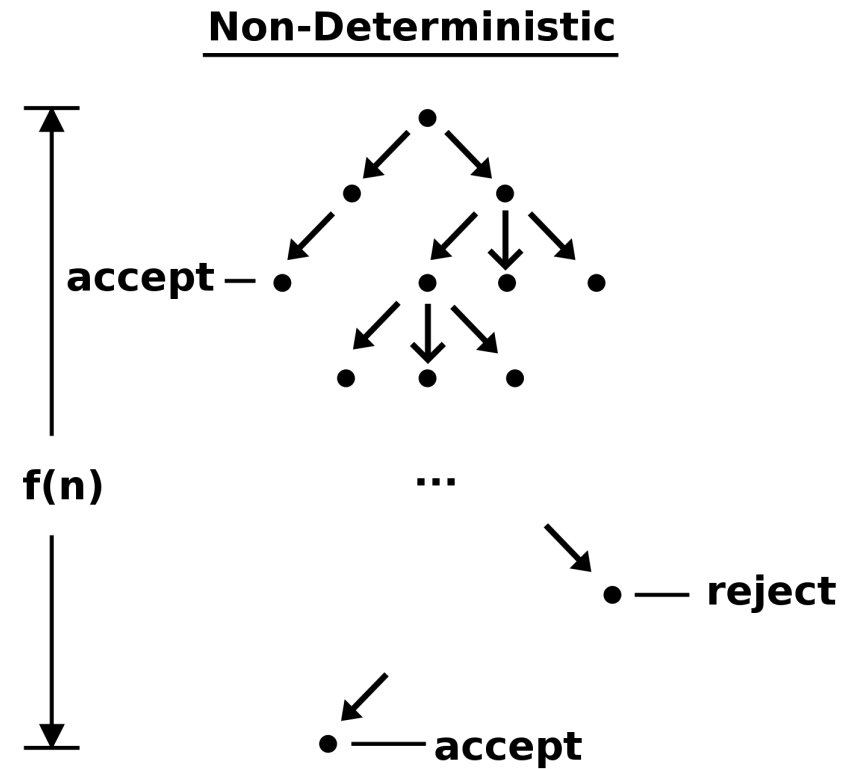
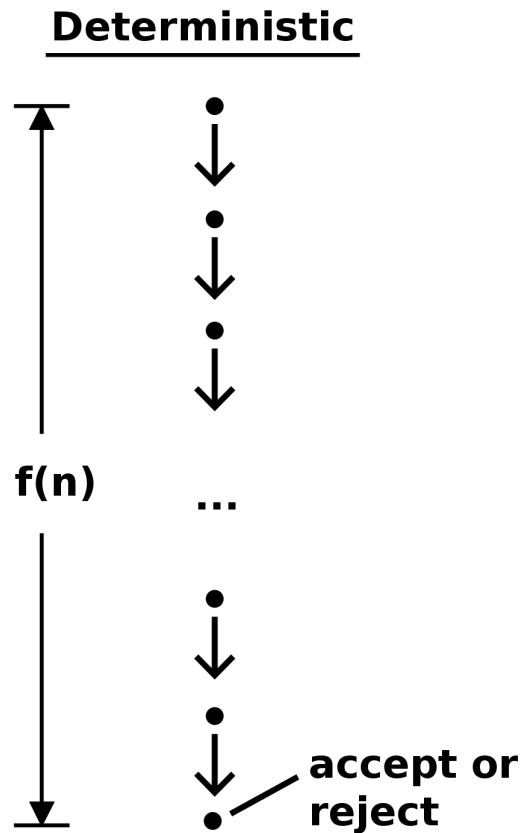
- A **complexity class** is a set of computational problems of related resource-based complexity. The two most commonly analyzed resources are time and memory.

- A complexity class is defined in terms of a type of computational problem, a model of computation, and a bounded resource like time or memory.
- Complexity classes consist of decision problems that are solvable with a Turing machine, and are differentiated by their time or space (memory) requirements.
- The class P is the set of decision problems solvable by a deterministic Turing machine in polynomial time.
- NP is the class of problems that are solvable by a nondeterministic Turing machine in polynomial time.
- Many complexity classes defined in terms of other types of problems (e.g. counting problems and function problems) and using other models of computation (e.g. probabilistic Turing machines, interactive proof systems, Boolean circuits, and quantum computers).



The Decision Problem

Deterministic and Non-Deterministic Turing Machines



So, why it is important to know all of this?

- Because you can decide how to attack a physical problem using computation
 - Selecting type of algorithm and possible design of the treatment from the mathematical representation (Remember the Big O)
 - Selecting the language and optimisation possibilities. (Or interpreters as Python)
 - Selecting the kind of computer to use (computer architecture characteristics)
 - Classical Von-Newman Computer
 - Non Von-Newman Computer (As a Quantum Computer)
 - Variations and Hybrid Computer (i.e. using multiple processors : CPUs, GPUs, XPU, DPUs, ASICs, etc.)



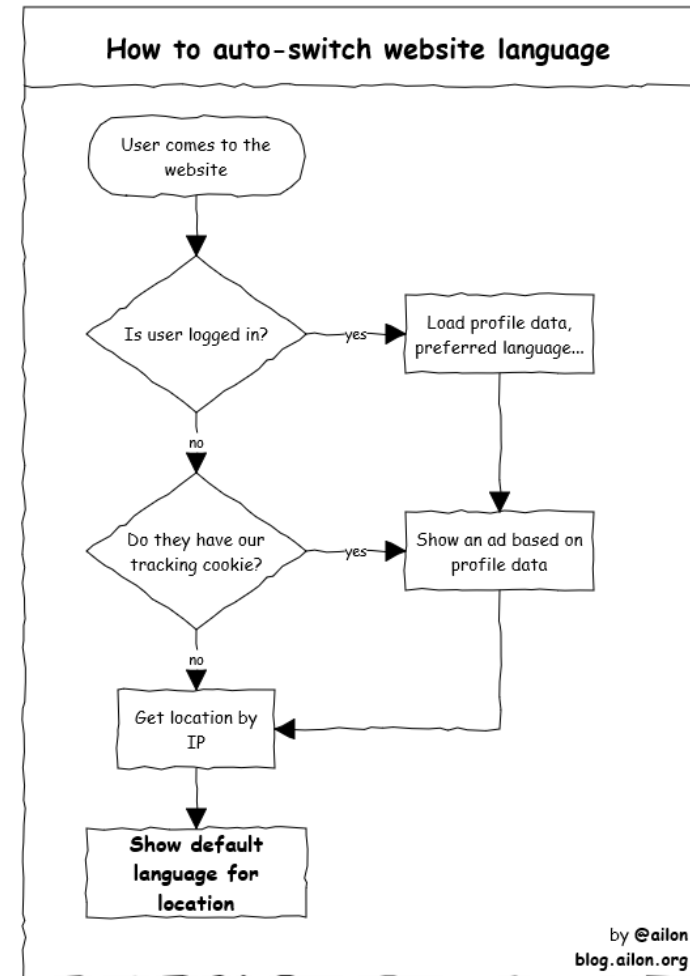
phd.stanford.edu/

- Selecting Programming Paradigms (Sequential, Parallel Computing (Shared Memory, Distributed Memory, Hybrid Memory))
- Because Big Problems need Smart Solutions

Now, time to work in Class (In teams)

1. The Simple Daily Problem

- Propose an algorithm (flowchart and pseudocode) for a simple daily task (i.e. walk to the classroom from the door of the building to your desktop, send a message by whatsapp...)
- Try to Analyze complexity and other characteristics (i.e. Number of steps, possible Big O, class of complexity)

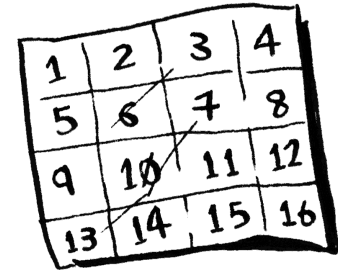


2. Visualizing different Big O run times

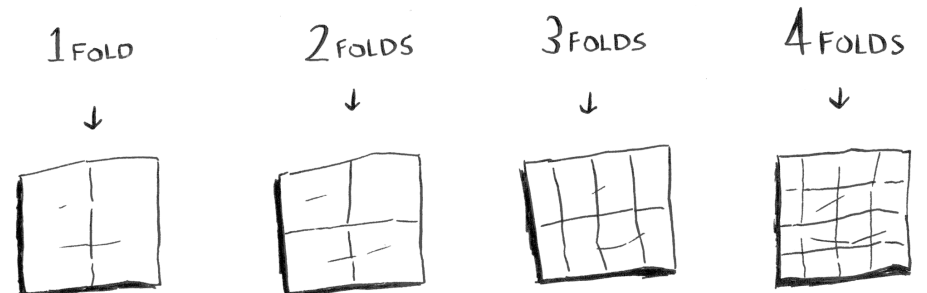
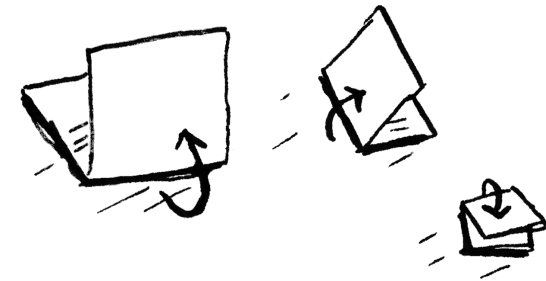
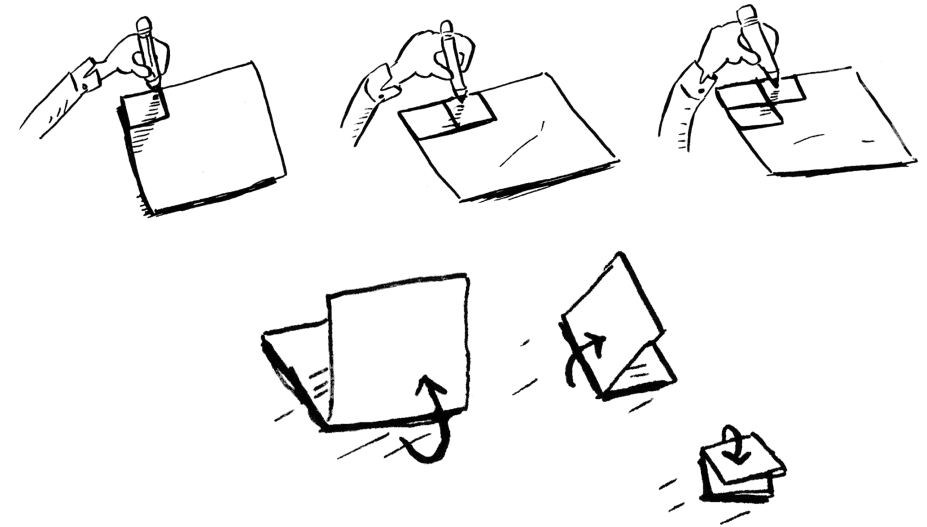
Take a Piece of paper and a pencil. Suppose you have to draw a grid of 16 boxes.

You have the possibility of two algorithms:

- **Algorithm 1:** Draw one box at time. How many operations will it take, drawing one box at time?
- **Algorithm 2:** Fold the paper, again and again, and again. Unfold it after four folds. How many operations will it take?
- Taking the Big O notation, what algorithm is linear and what is logarithmic?



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



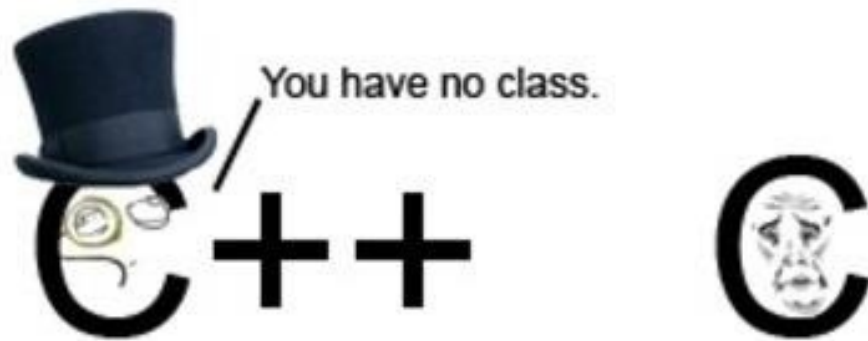
From: https://github.com/egonSchiele/grokking_algorithms

Coding is a process

- Know the Problem
- Know and Manage Language Programming Elements and Advantages
 - And use them!
- Know Computer Architecture and Its Real Performance (Expected)
 - And Exploit them!
- Good Algorithms produce Good Codes
 - Good Practices
- Experience (and continuous auto-learning)



C or C++ ? (Or any language)



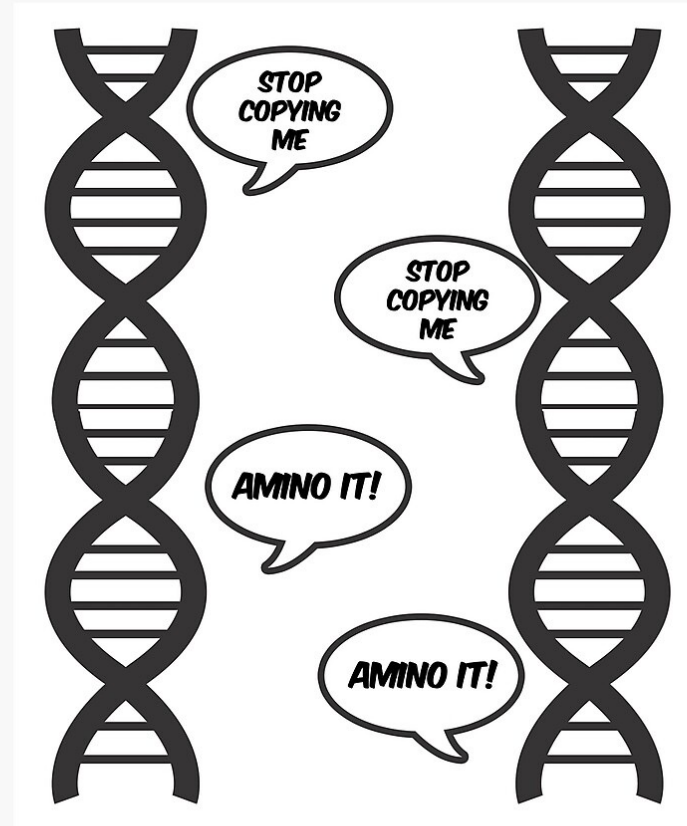
Depending of the oriented programming

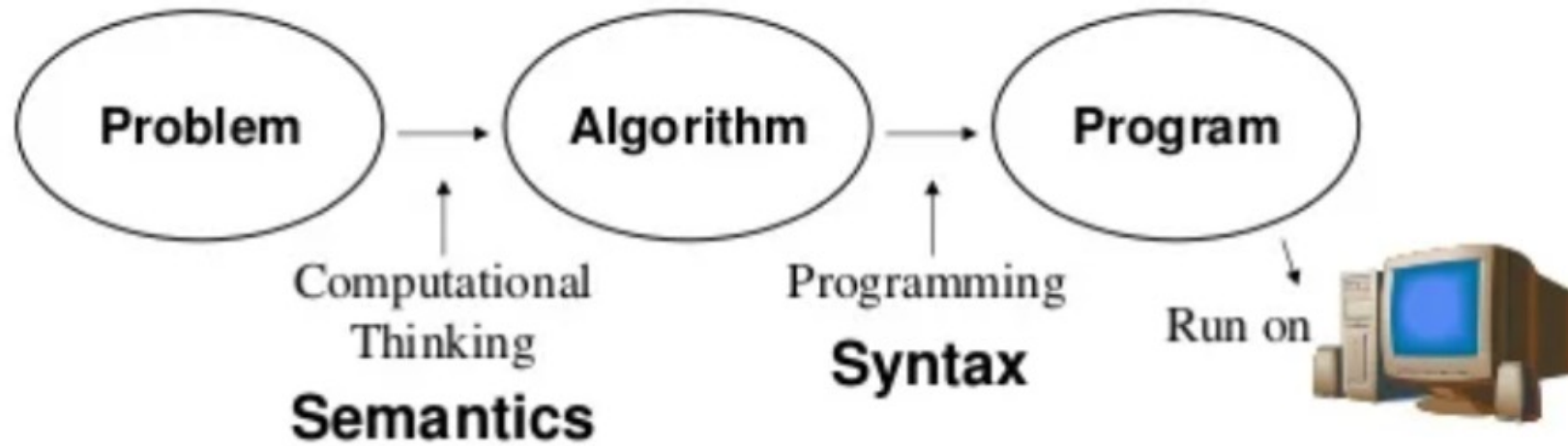
Programming paradigm and oriented development is a process (scientific, technical and engineering).

- You need to see the requirements (or expected outcomes)
- Programability (Remember, the idea is that you're a scientists, so your time is to make science)
- Skills
- Platforms (not only computer machines), runtime.
- Likes and comfort!
 - Not always, the popular is good

Special Recommendations about Copying

- Programming in real life, copying is strongly encouraged. (The idea is not to reinvent the wheel (and not waste time getting it wrong)
 - Copying saves time;
 - Copying avoids typing mistakes;
 - Copying allows you to focus on your new programming challenges.





SYNTAX & SEMANTIC (1/2)

- Programming language enforces a set of rules, symbols and special words used to construct a program.
- A set of rules that precisely state the *validity of the instructions* used to construct a program is called syntax or 'grammar' else syntax error will be generated.
- The correctness of the instructions used to write any program is called semantics or *correct meaning*.
- These set of rules for instructions validity and correctness are monitored by the compilers.
- Semantically one but can have many syntaxes.

SYNTAX & SEMANTIC (2/2)

- e.g.

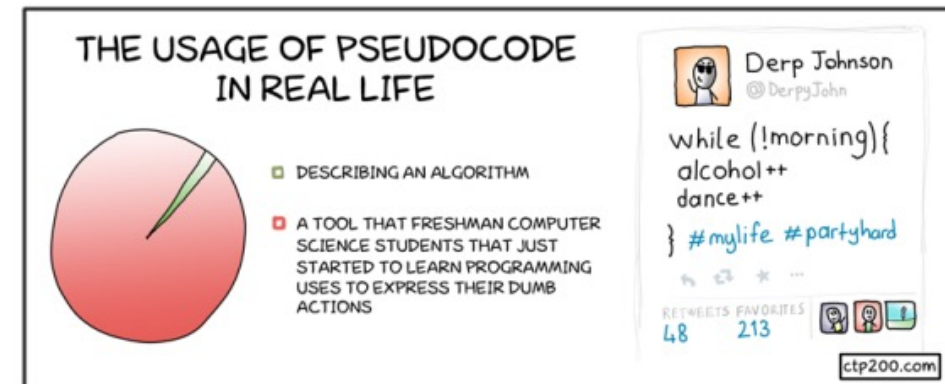
To add an integer to a variable q and store the result in q (semantic), syntactically (correct), we can write:

$q = q + 3$; or $q += 3$;

- *Pseudocode* - an informal high-level description of the operating principle of a computer program or other algorithm.
- Uses the structural conventions of a programming language, but is intended for human reading rather than machine reading.

(Remember) PSEUDOCODE & ALGORITHM (1/3)

- An informal high-level description of a computer program or algorithm operating principle.
- An *algorithm* is merely the sequence of steps taken to solve a problem which are normally a sequence, selection, iteration and a case-type statement.
- Algorithm is a procedure for solving a problem - actions to be executed and the order in which those actions are to be executed.
- Every algorithm may have different number line of code, different repetition loops, different execution speeds etc.



(Remember) PSEUDOCODE & ALGORITHM (2/3)

- But all the program have similar purpose: to sort the given unsorted integers in ascending order.
- Pseudocode uses programming language's structural conventions , intended for human rather than machine reading.
- helps programmers develop algorithms.

algorithm

noun

Word used by programmers when they do not want to explain what they did.

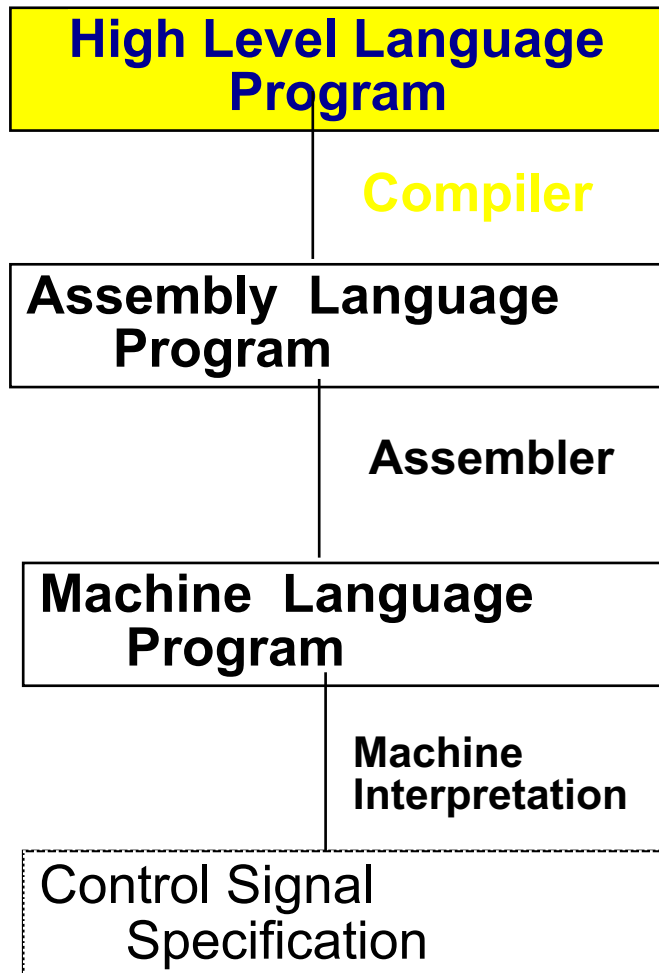
(Remember) PSEUDOCODE & ALGORITHM (3/3)

```
Set sum to zero
Set grade counter to one
While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the sum
Set the class average to the sum divided by ten
Print the class average.
```

```
IF HoursWorked > NormalMax THEN
    Display overtime message
ELSE
    Display regular time message
ENDIF
```

```
SET total to zero
REPEAT
    READ Temperature
    IF Temperature > Freezing THEN
        INCREMENT total
    END IF
UNTIL Temperature < zero
Print total
```


Levels of Representation



```
int main()
```

```
{
```

```
    temp = v[k];
```

```
    v[k] = v[k+1];
```

```
    v[k+1] = temp;
```

```
}
```

```
gcc program.c -o program
```

```
lw  $15,    0($2)
```

```
lw  $16,    4($2)
```

```
sw  $16,    0($2)
```

```
sw  $15,    4($2)
```

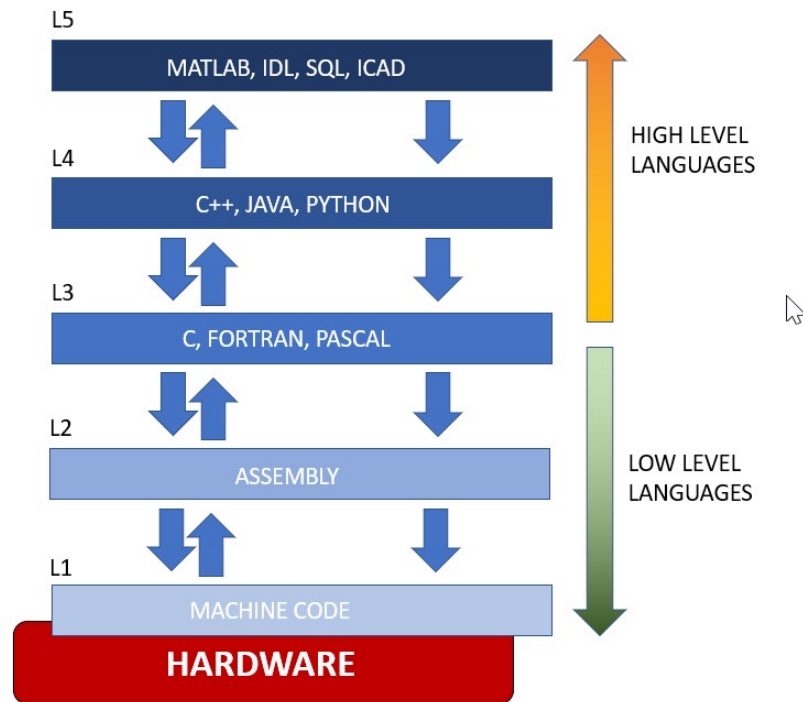
```
0000 1001 1100 0110 1010 1111 0101 1000
```

```
1010 1111 0101 1000 0000 1001 1100 0110
```

```
1100 0110 1010 1111 0101 1000 0000 1001
```

```
0101 1000 0000 1001 1100 0110 1010 1111
```

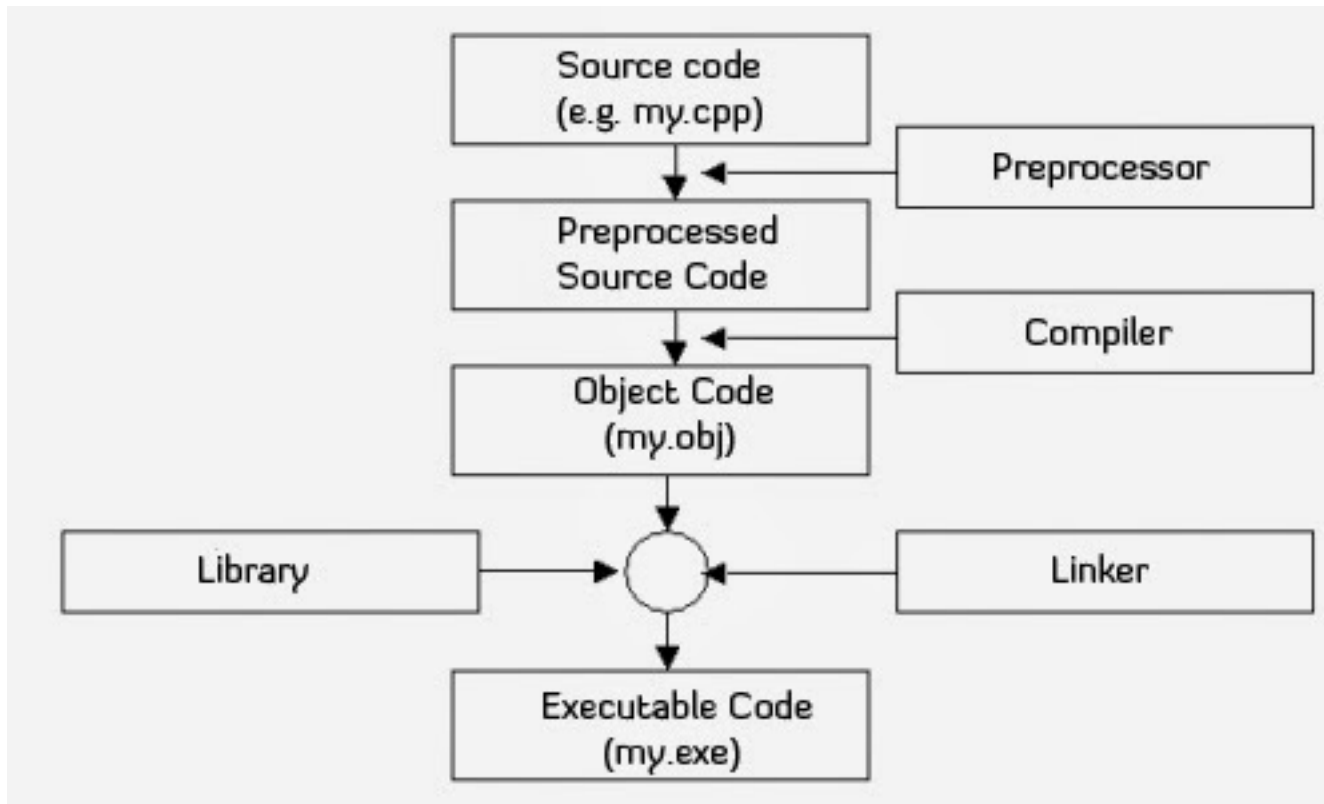
High Level Programming Languages



- The primary purpose of a high-level language is to permit more direct expression of a programmer's design.
- High Level code modules can be designed to « plug » together piece by piece, allowing large programs to be built out of small, comprehensible parts.
- Ideally, a program written in a high-level language may be ported to a different machine and run without change.

Compiled code is not the only way to execute high-level programs. An alternative is to translate the program on-the-fly using an interpreter program (e.g., Matlab, Python, etc).

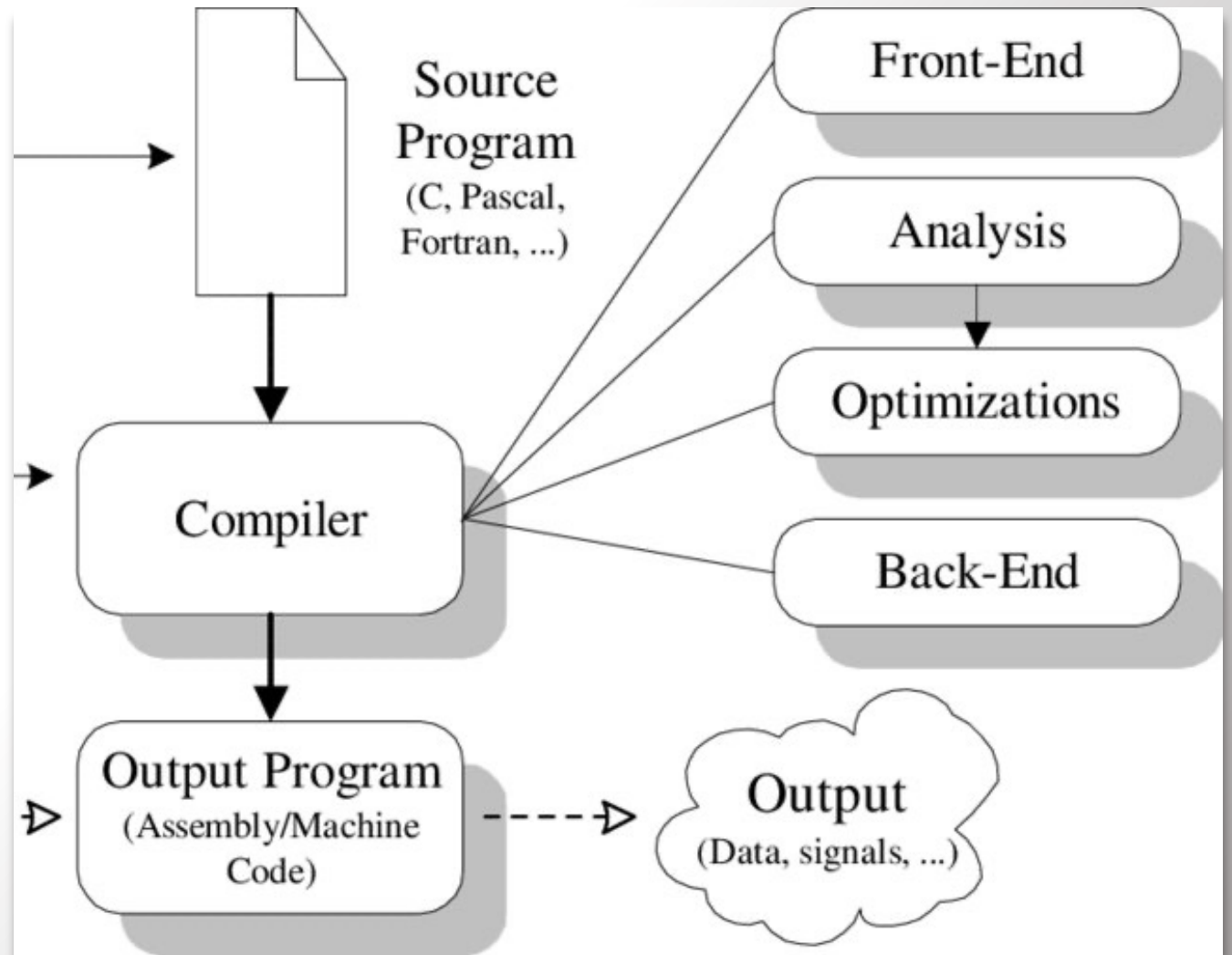
From pseudocode to executable code



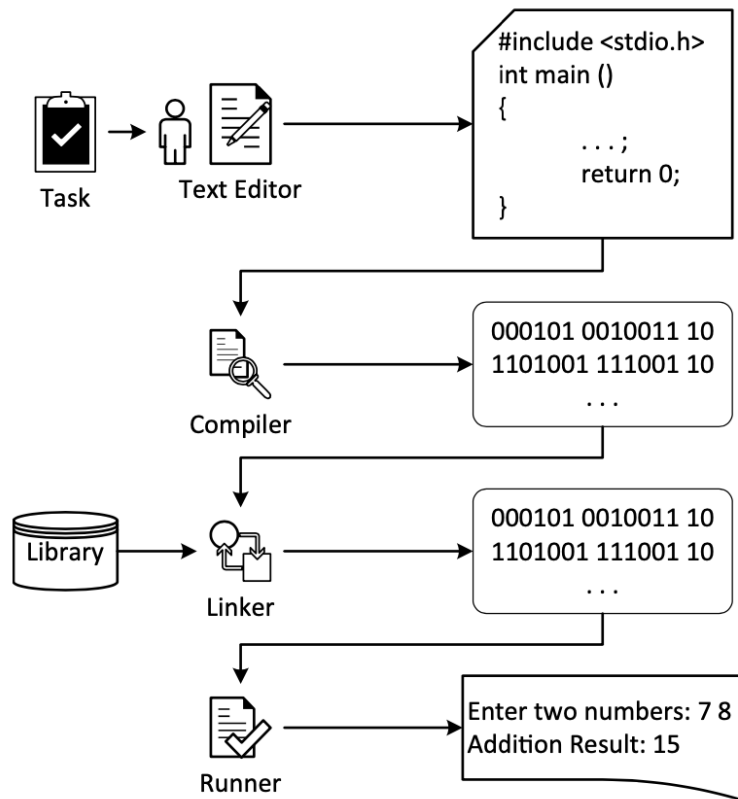
- To produce executable code from a such program, it is translated to machine specific assembler language by a compiler program, wich is the converted to machine code by an assembler.
- An executable code is a special type of file that contains machine instructions (ones and zeros), and running this file causes the computer to perform those instructions.

Compiling

- Compiling is the process of turning a program files into an executable.
- The compilation process involves stages and utilizes different 'tools' or other programs such as a preprocessor, compiler, assembler, and linker in order to create a final executable.



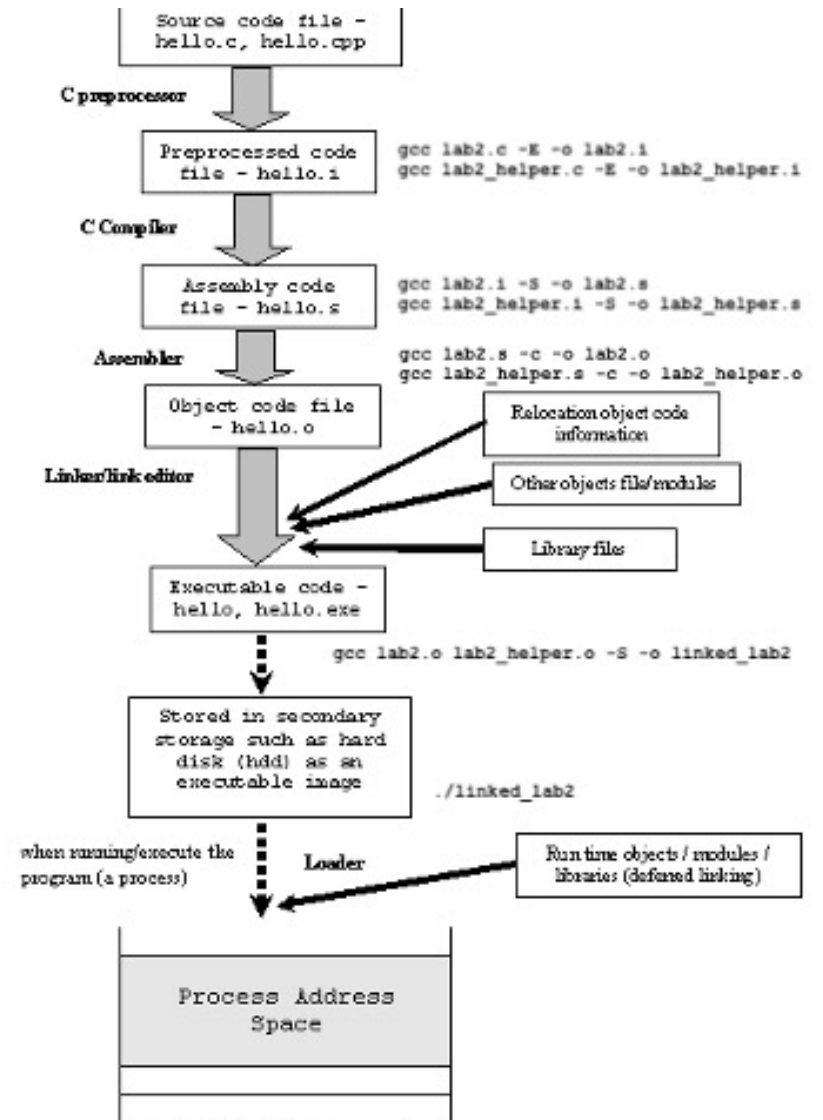
Compiling and Running



1. You build a code/program file using an specific identifier (i.e., **Myfile.c** to c code or **Myfile.cpp** to c++ code)
2. Following the syntaxis and rules you “compile” your code, using a compiler (to see after) (i.e., **gcc Myfile.c -o MyExe**, using gcc compiler)
3. Finally, You use the executable file to execute your code (i.e., in console mode: **./MyExe**)

Compiling Stages (Traditional Vision)

1. **Preprocessing** is the first pass of any compilation. It removes comments, expands include-files and macros, and processes conditional compilation instructions. This can be output as a .i file.
2. **Compilation** is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code (hello.s). Assembly language is a low-level programming language (even lower than C) that is still human-readable but consists of mnemonic instructions that have strong correspondence to machine instructions.
3. **Assembly** is the third stage of compilation. It takes the assembly source code and produces an object file hello.o, which contains actual machine instructions and symbols (e.g., function names) that are no longer human-readable since they are in bits.
4. **Linking** is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).



More Stages



- Debugging and Profiling
 - Debugging: Intended to determine correctness issues.
 - Can occurs during compiling process.
 - Profiling: Intended to determine diagnostic/Performance issues.
 - Can occurs after compiling process, i.e., after execution or deployment process.
- It exists technics, tools, programs and intuition.

First Good Practice: Order

- A program consists of functions and variables. A function contains statements that specify the computing operations to be done, and variables store values used during the computation.
- A program is structured, actually, the program is consisting of 6 main sections.
- Structured coding is a good practice of good programmers. (And allows to develop and maintain good codes).

```
/*Documentation Section:
  Program Name: program to find the area of circle
  Author: Rumman Ansari
  Date : 12/01/2013
*/

#include"stdio.h" //Link section
#include"conio.h" //Link section

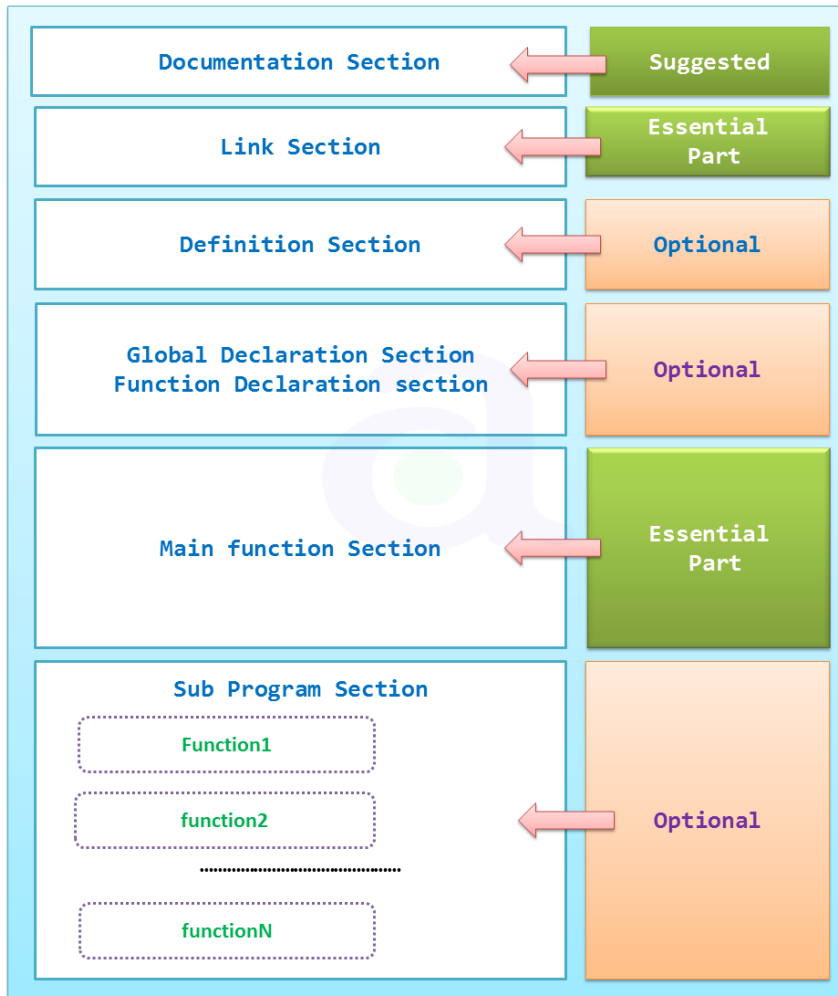
#define PI 3.14 //Definition section

float area; //Global declaration section
void message(); //function prototype declaration section

void main()
{
    float r; //Declaration part
    printf("Enter the radius \n"); //Executable part
    scanf("%f",&r);
    area=PI*r*r; // Calculation Part
    printf("Area of the circle=%f \n",area);
    message(); // Function Calling
}

// Sub function
void message()
{
    printf("This Sub Function \n");
    printf("we can take more Sub Function \n");
}
```


More about the structure



```
/*Documentation Section:
Program Name: program to find the area of circle
Author: Rumman Ansari
Date : 12/01/2013
*/

#include"stdio.h" //Link section
#include"conio.h" //Link section

#define PI 3.14 //Definition section

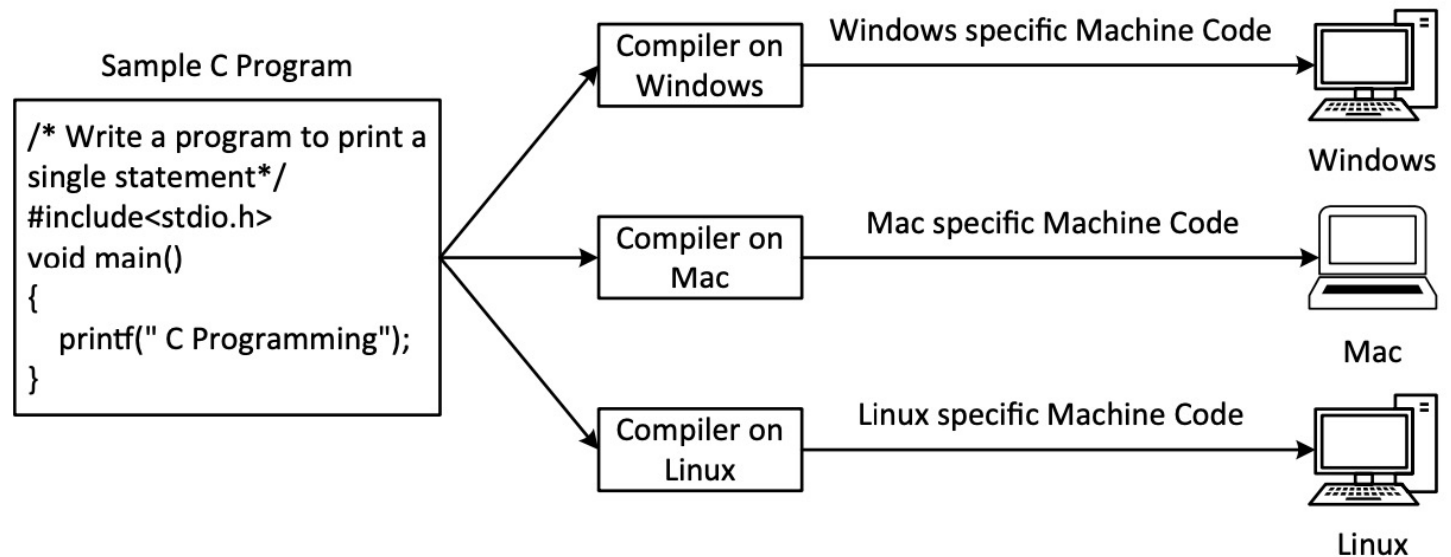
float area; //Global declaration section
void message(); //function prototype declaration section

void main()
{
    float r; //Declaration part
    printf("Enter the radius \n"); //Executable part
    scanf("%f",&r);
    area=PI*r*r; // Calculation Part
    printf("Area of the circle=%f \n",area);
    message(); // Function Calling
}

// Sub function
void message()
{
    printf("This Sub Function \n");
    printf("we can take more Sub Function \n");
}
```

About Compilers

- A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses.
- The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.
- Depending of the machine, Environment-specific machine code generation by different compilers.



From: **C Programming Learn to Code** Sisir Kumar Jena

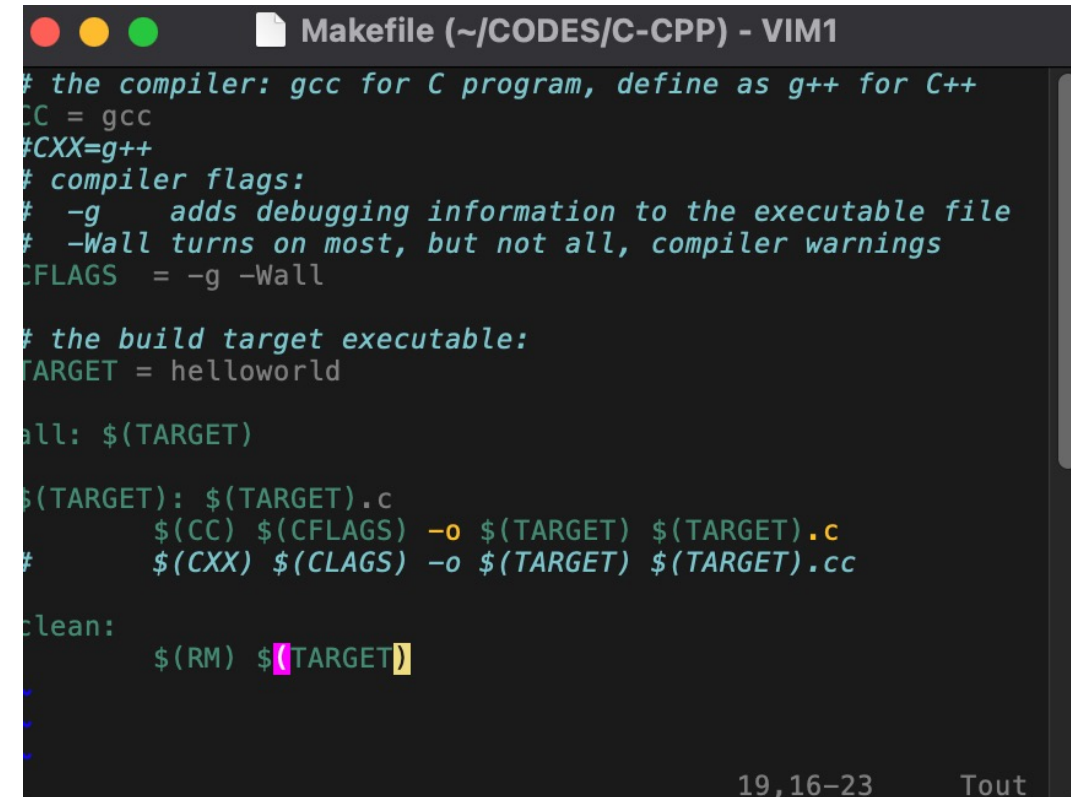
Some C Compilers and Standards

Compiler	Standard
Amsterdam Compiler Kit Clang, using LLVM backend	C K&R and C89/90
GCC	C89, C90, C99, and C11 C89/90, C99, and C11
HP C/ANSI C compiler Microsoft Visual C++	C89 and C99
Pelles C	C89/90 and C99
Vbcc	C99 and C11 (Windows only) C89/90 and C99
Tiny C Compiler	C89/90 and some C99
Intel C Compiler	C89, C90, C99, and C11 C89/90, C99, and C11

More in: https://en.wikipedia.org/wiki/List_of_compilers#C_compilers

2 Good Practice: Use Makefiles

- The `make` allows to define personal rules to compiling your C / C++ code using the warning flags automatically. (Also, you can add other languages primitives)
- Make is a build automation tool that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program.



```
Makefile (~/CODES/C-CPP) - VIM1
# the compiler: gcc for C program, define as g++ for C++
CC = gcc
CXX=g++
# compiler flags:
# -g adds debugging information to the executable file
# -Wall turns on most, but not all, compiler warnings
CFLAGS = -g -Wall

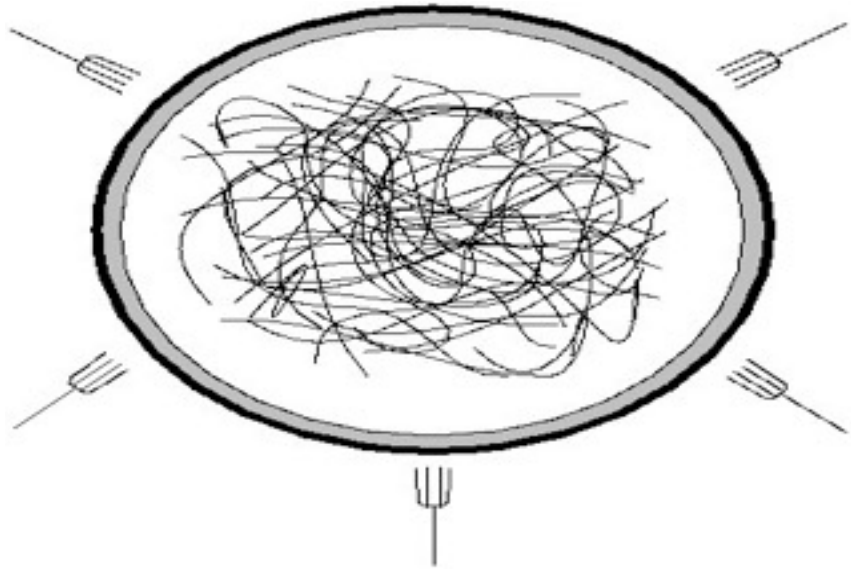
# the build target executable:
TARGET = helloworld

all: $(TARGET)

$(TARGET): $(TARGET).c
    $(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c
#    $(CXX) $(CLAGS) -o $(TARGET) $(TARGET).cc

clean:
    $(RM) $(TARGET)
```

About Parallelism

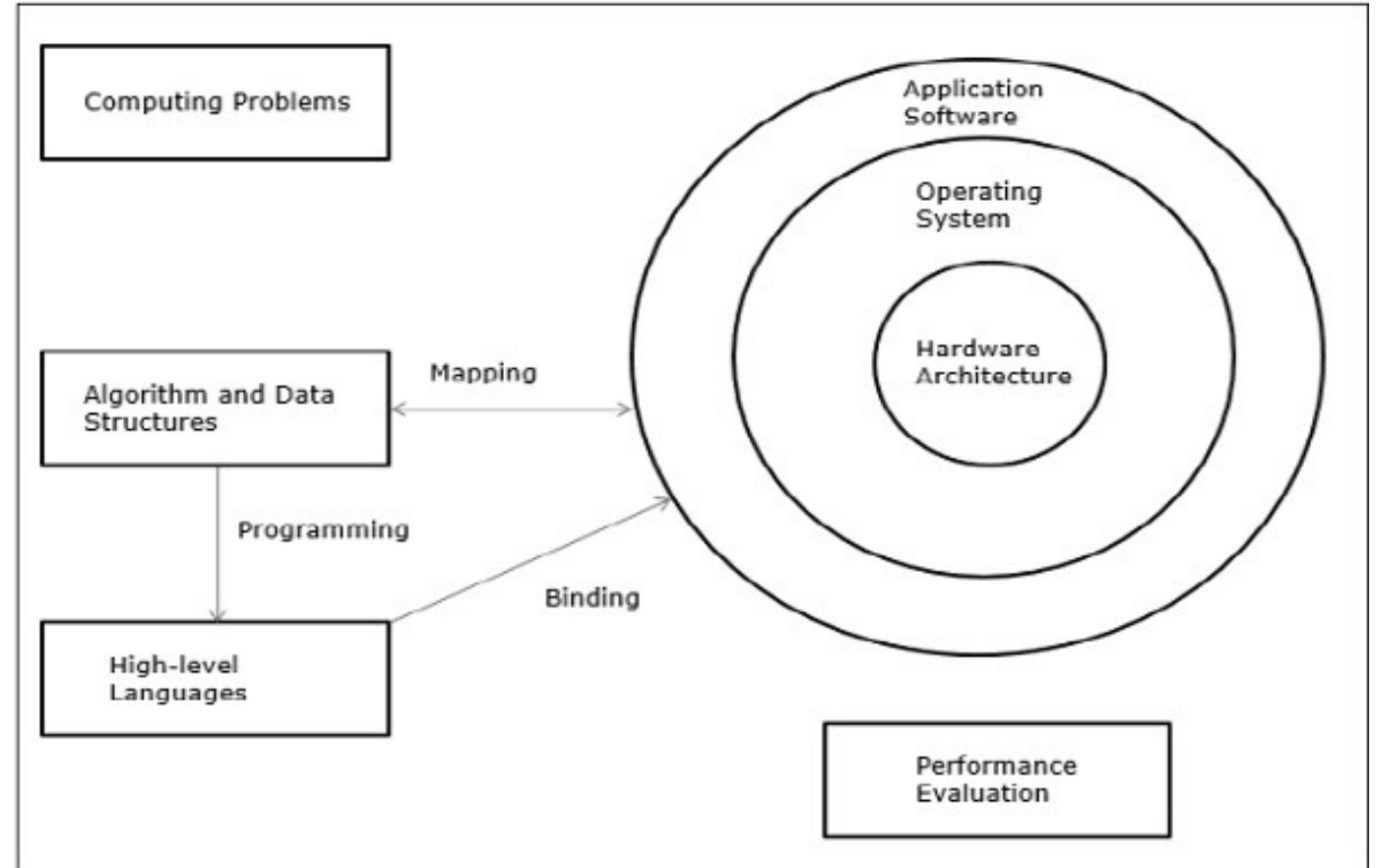


- **Concurrency** is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other.

- **Implicit parallelism** is a characteristic of a programming language that allows a compiler or interpreter to automatically exploit the parallelism inherent to the computations expressed by some of the language's constructs.
- **Explicit parallelism** is the representation of concurrent computations by means of primitives in the form of special-purpose directives or function calls.

Elements of Parallelism

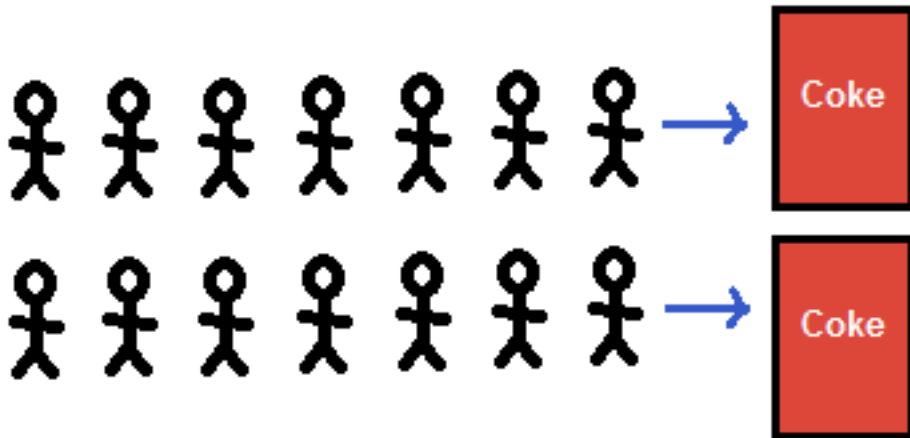
1. Computing Problems
 - Numerical (Intensive Computing, Large Data Sets)
 - Logical (AI Problems)
2. Parallel Algorithms and Data Structures
 - Special Algorithms (Numerical, Symbolic)
 - Data Structures (Dependency Analysis)
 - Interdisciplinary Action (Due to the Computing Problems)
3. System Software Support
 - High Level Languages (HLL)
 - Assemblers, Linkers, Loaders
 - Models Programming
 - Portable Parallel Programming Directives and Libraries
 - User Interfaces and Tools
4. Compiler Support
 - Implicit Parallelism Approach
 - Parallelizing Compiler
 - Source Codes
 - Explicit parallelism Approach
 - Programmer Explicitly
 - Sequential Compilers, Low Level Libraries
 - Concurrent Compilers (HLL)
 - Concurrency Preserving Compiler
5. Parallel Hardware Architecture
 - Processors
 - Memory
 - Network and I/O
 - Storage



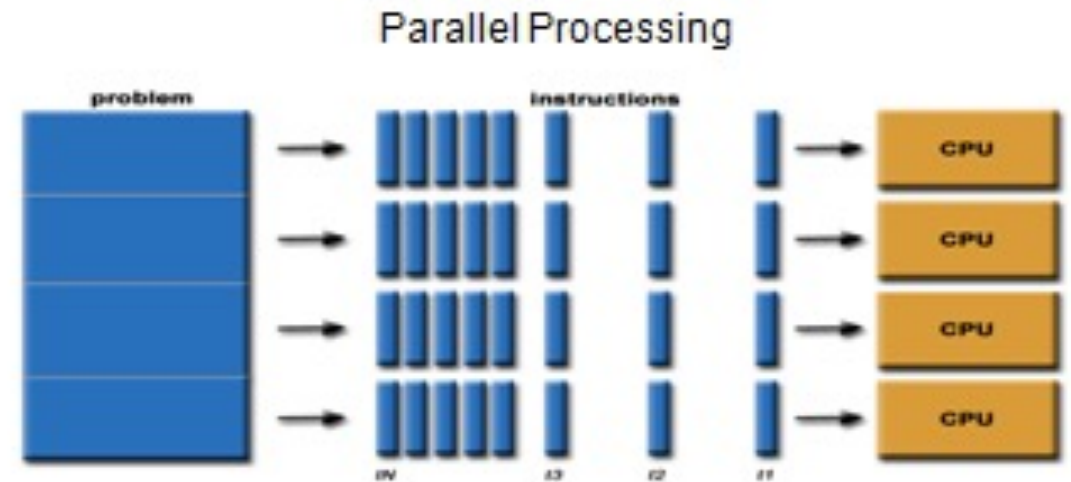
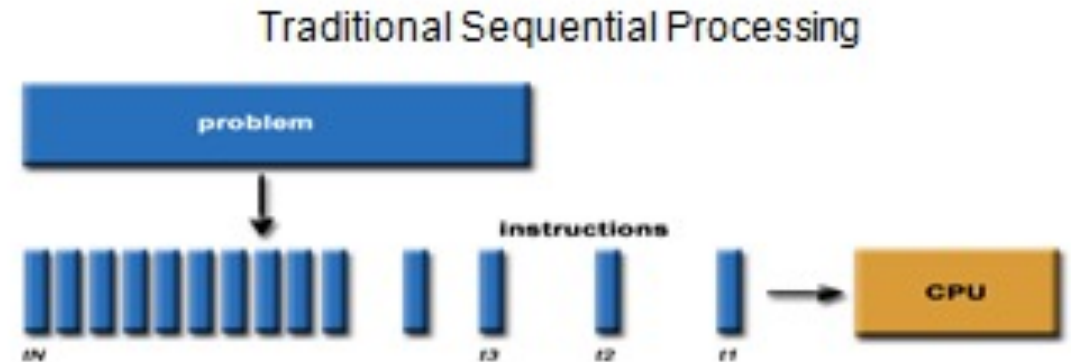
Thinking in Parallel (computing) – The Typical Visions



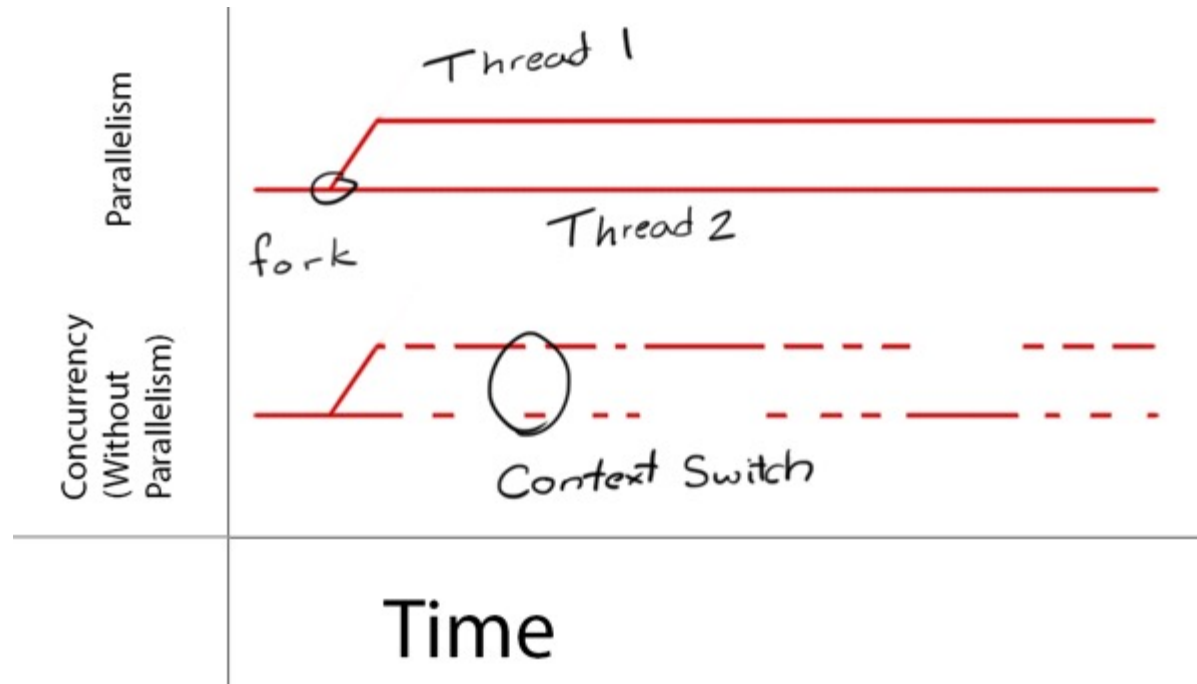
Concurrent: 2 queues, 1 vending machine



Parallel: 2 queues, 2 vending machines



Concurrency vs Concurrency/Parallelism Behavior



Non Shared Processing Ressources (However the Memory...)

Switching

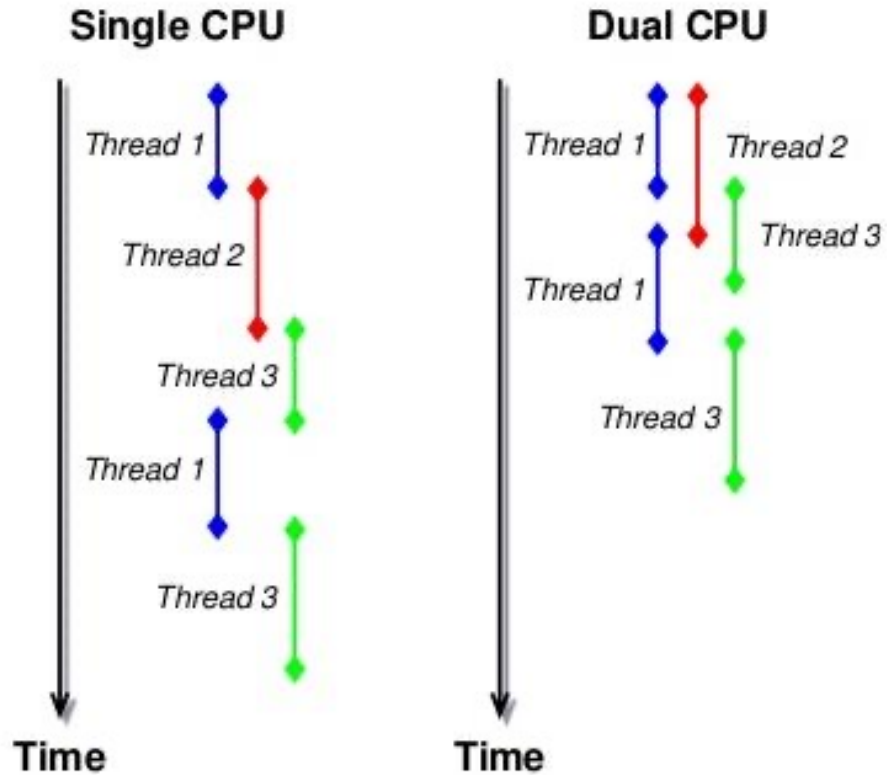
Parallel Threads (Multitasking, Multithreading)

Shared Processing Ressources

Switching

Non Parallel Threads (Non Multitasking, Yes Multithreading)

Concurrency vs Concurrency/Parallelism Example



Single System

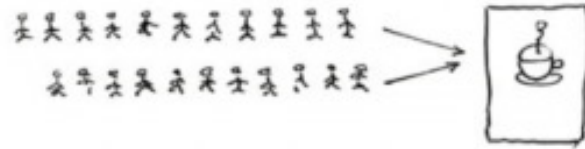
- Multiple Threads in Runtime
- Almost Synchronization Strategies
- Memory Allocation

Dual System

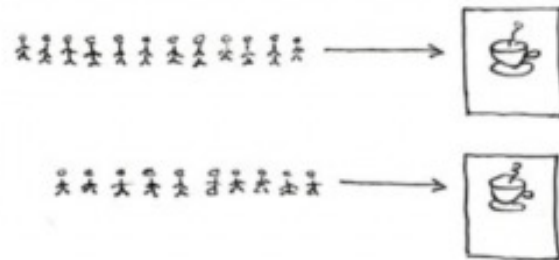
- Multiple Parallel Threads in Runtime
- Strategies to Parallelism following models (PRAM, LogP, etc) addressed to exploit memory and overhead reduction

CONCURRENCY | PARALLELISM

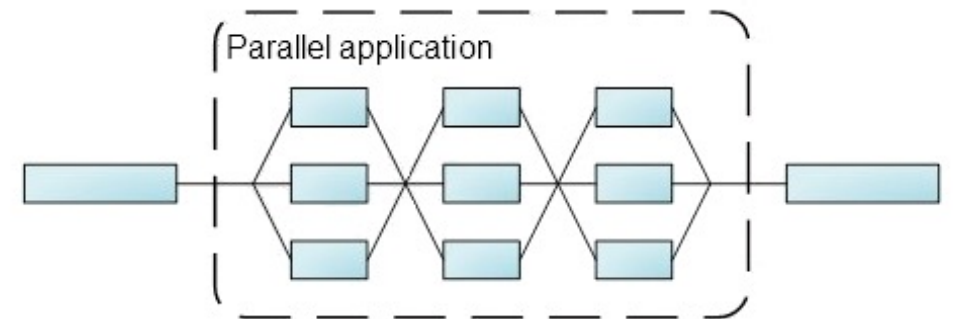
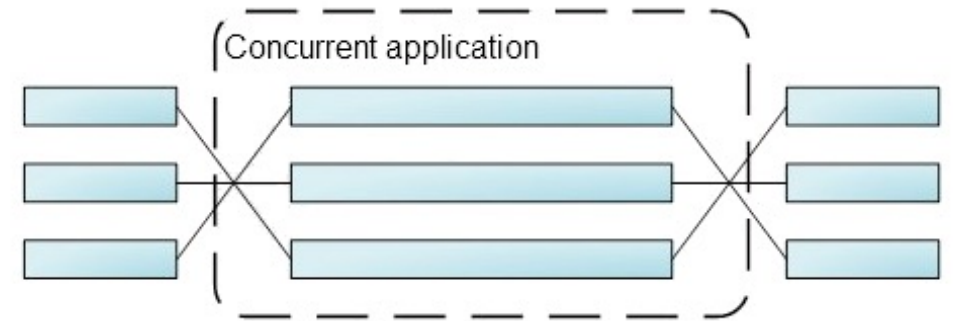
Concurrent = Two Queues One Coffee Machine



Parallel = Two Queues Two Coffee Machines



© Joe Armstrong 2018



From J. Armstrong Notes: <http://joearms.github.io/2013/04/05/concurrent-and-parallel-programming.html>

Any Parallel System is concurrent: Simultaneous Processing, Parallel but limited resources.

Advantages of Concurrency

- **Concurrent processes can reduce duplication in code.**
- **The overall runtime of the algorithm can be significantly reduced.**
- **More real-world problems can be solved than with sequential algorithms alone.**
- **Redundancy can make systems more reliable.**

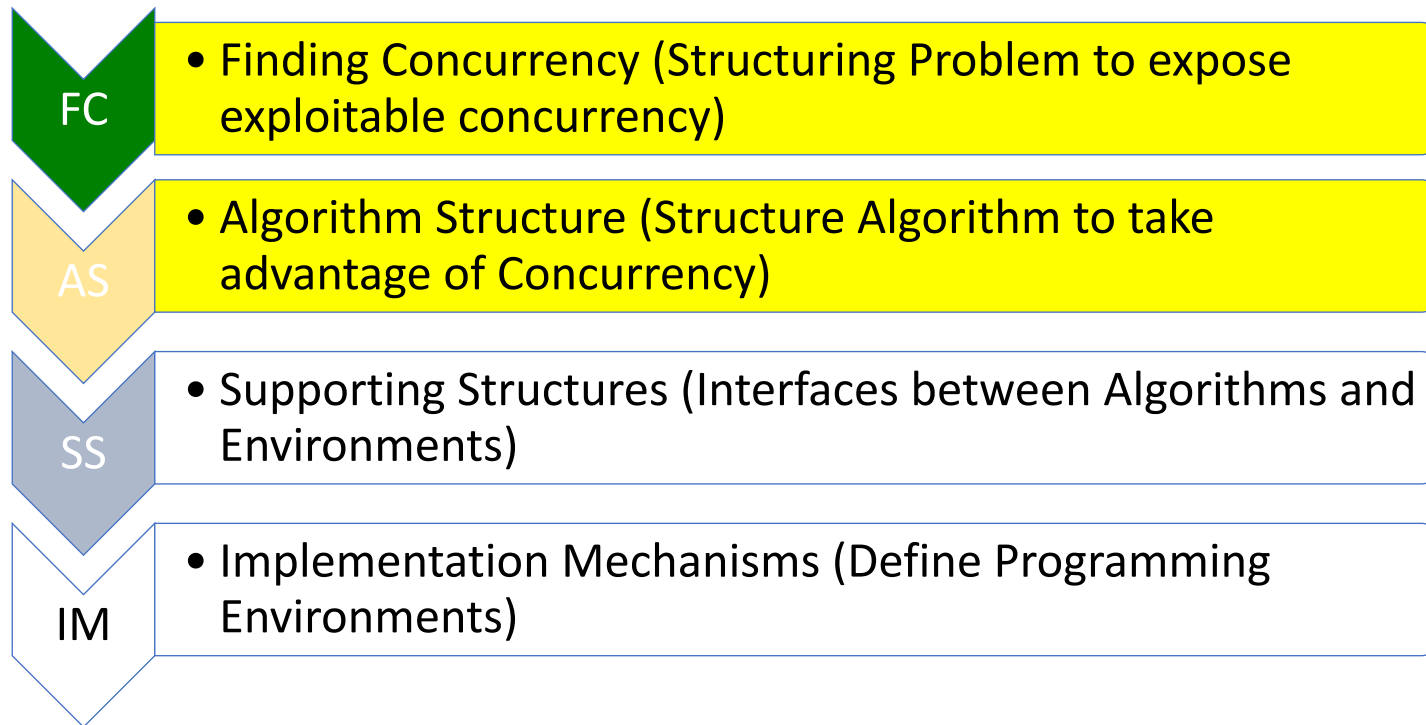
Disadvantages of Concurrency

- **Runtime is not always reduced, so careful planning is required**
- **Concurrent algorithms can be more complex than sequential algorithms**
- **Shared data can be corrupted**
- **Communications between tasks is needed**

3. Good Practice: Follow a workflow for Concurrent Programming

- **Analysis**
- Identify Possible Concurrency
 - Hotspot: Any partition of the code that has a significant amount of activity
 - Time spent, Independence of the code...
- **Design and Implementation**
- Threading the algorithm
- **Tests of Correctness**
- Detecting and Fixing Threading Errors
- **Tune of Performance**
- Removing Performance Bottlenecks
 - Logical errors, contention, synchronization errors, imbalance, excessive overhead
 - Tuning Performance Problems in the code (tuning cycles)

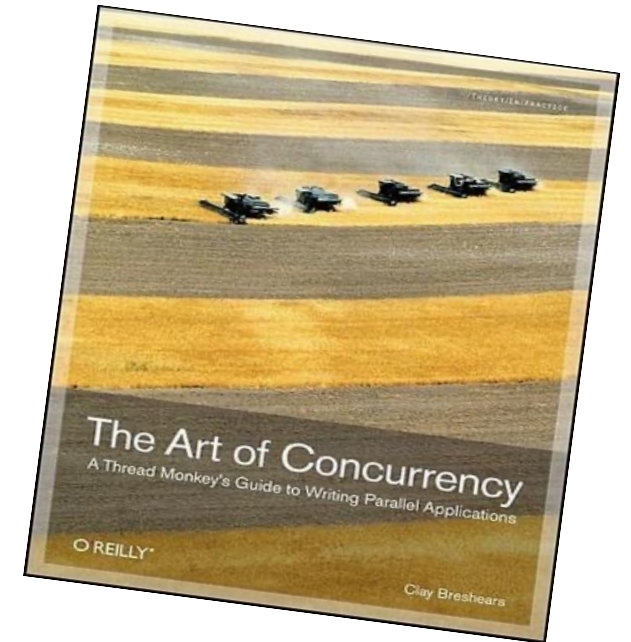
Design Spaces of Parallel Programming*



• Patterns for Parallel Programming, Timoty Mattson, Beverly A. Sanders and Berna L. Massingill, Software Pattern Series, Addison-Wesley 2004

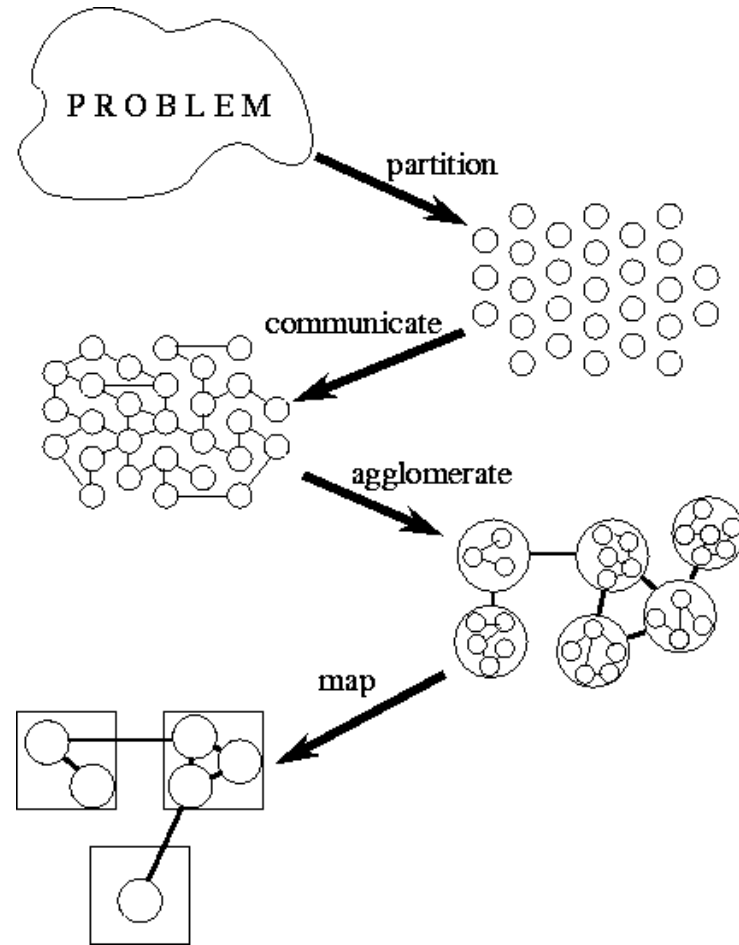
Parallel Computing

- Parallel Computing exploit Concurrency
 - In “system” terms, concurrency exists when a problem can be decomposed in sub problems that can safely executed at same time (in other words, concurrently)



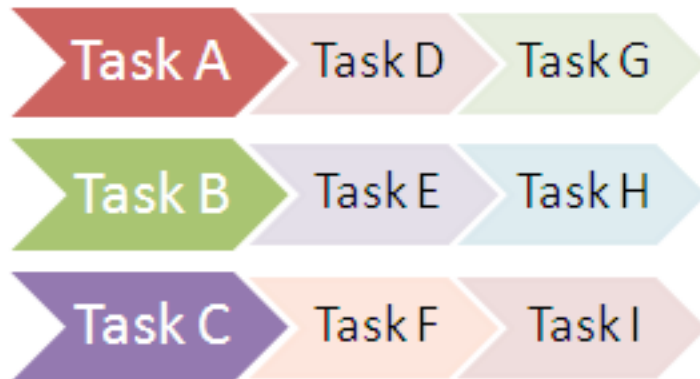
<https://ignorelist.files.wordpress.com/2012/01/the-art-of-concurrency.pdf>

Traditional Way

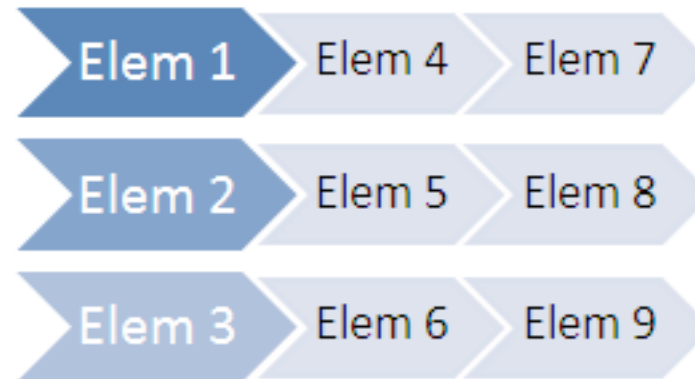


Descomposition

Task Parallelism



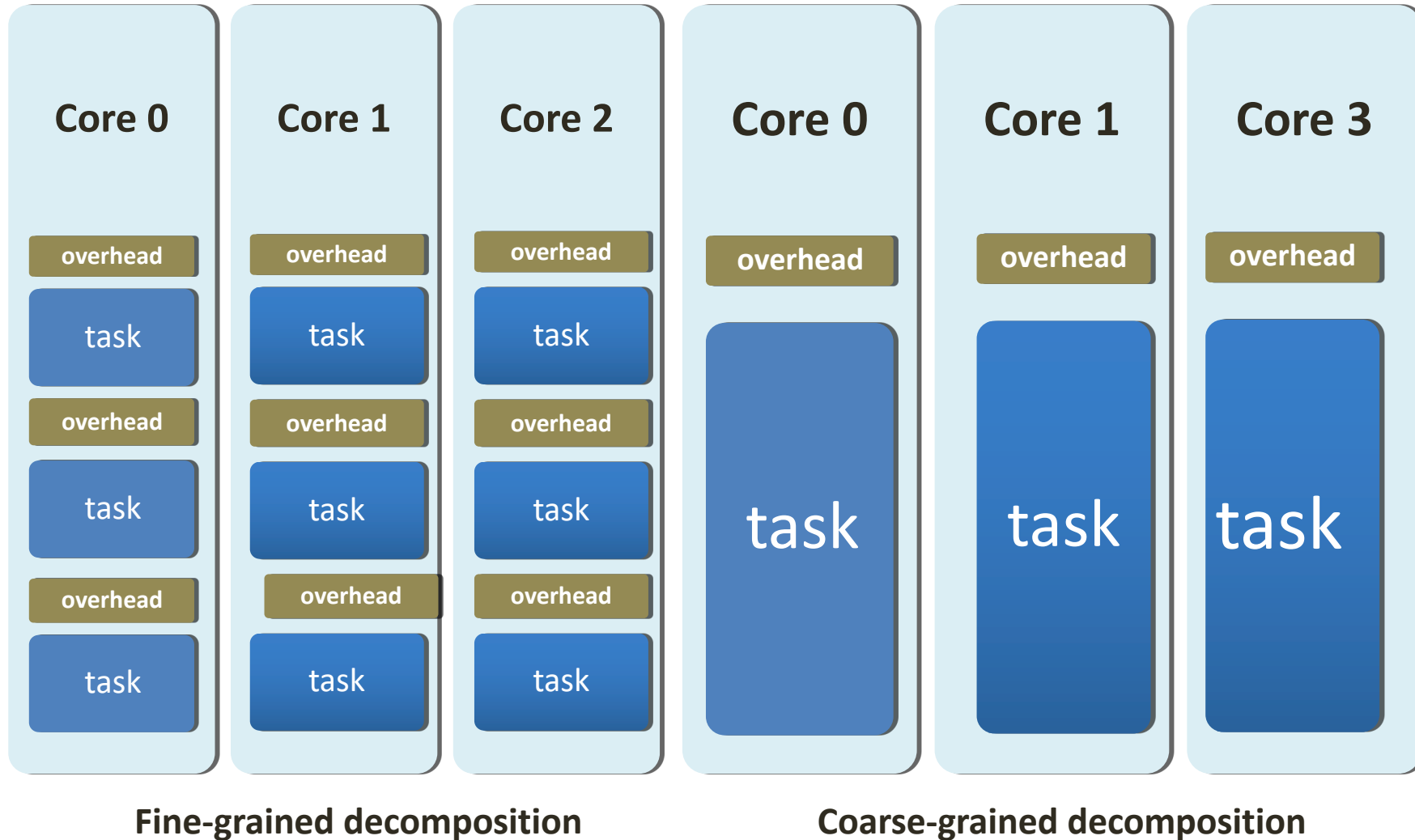
Data Parallelism



Tasks Decomposition : Task Parallelism

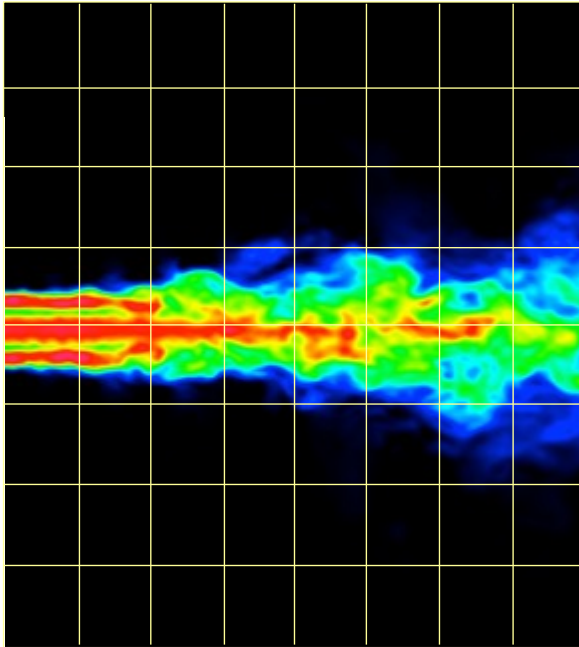
Data Decomposition: Data Parallelism /Geometric
Parallelism

Task Granularity



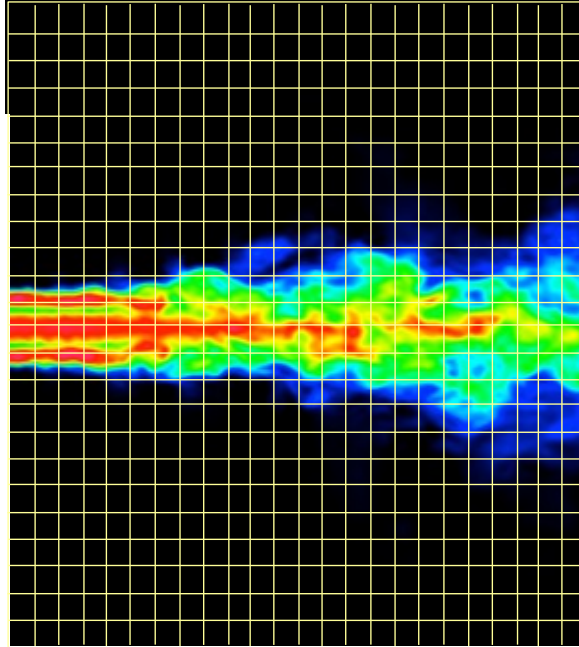
Granularity in Implementations

Coarse grid



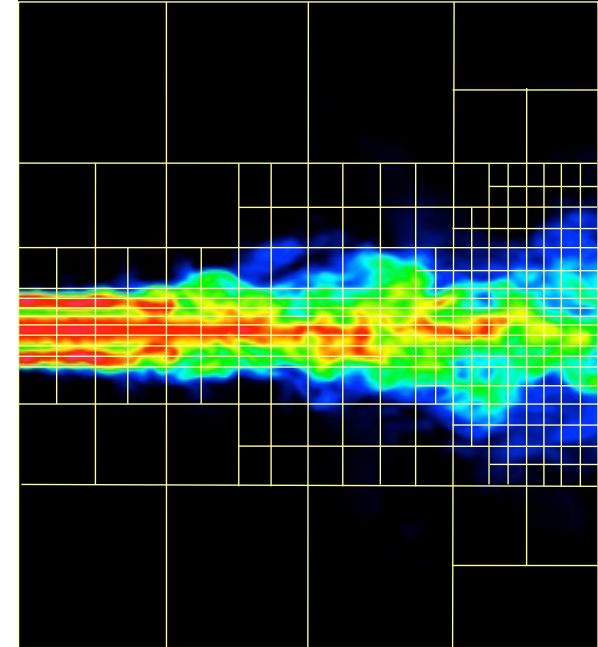
Higher Performance
Lower Accuracy
(Using Nodes)

Fine grid



Lower Performance
Higher Accuracy
(Using Processors)

Dynamic grid

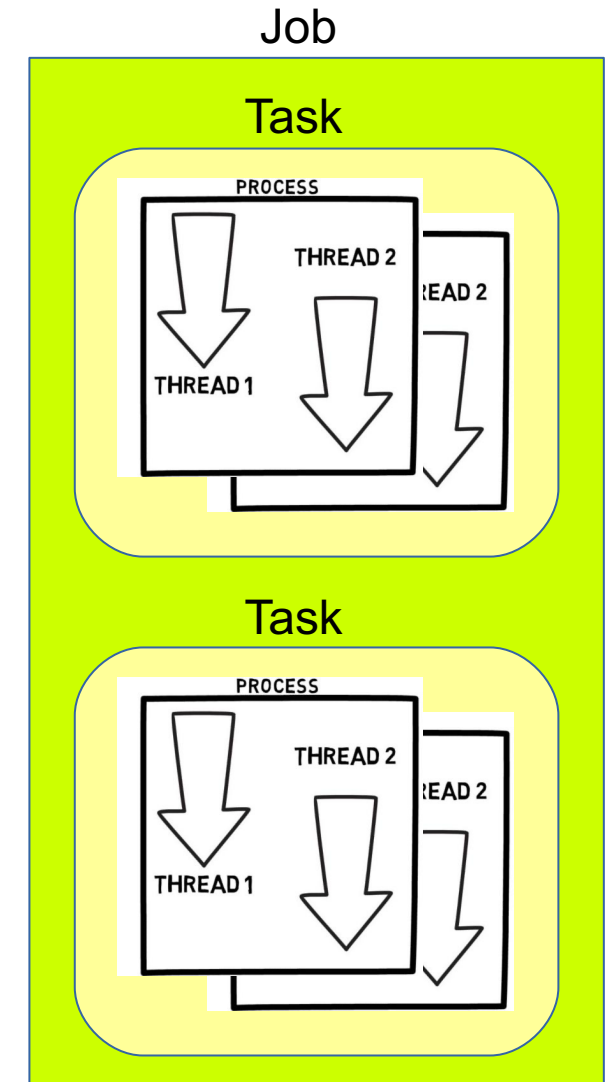


*Target performance where
accuracy is required
(Using Processors and
Nodes)*

Task Decomposition Considerations

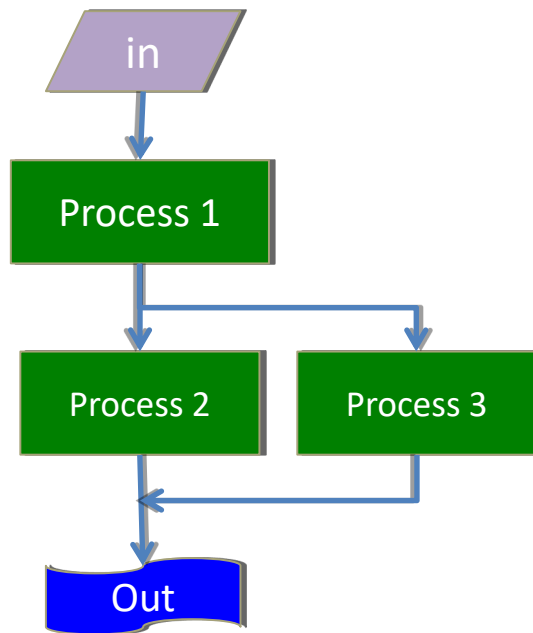
- What are the tasks and how are defined?
- What are the dependencies between task and how can they be satisfied?
- How are the task assigned to threads?

Tasks must be assigned to threads for execution

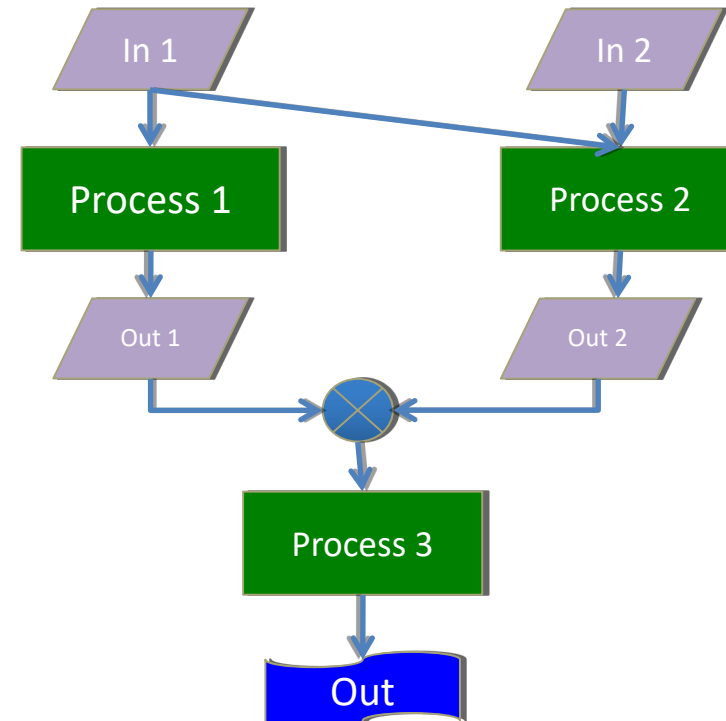


Task Dependencies

Order Dependency



Data Dependency



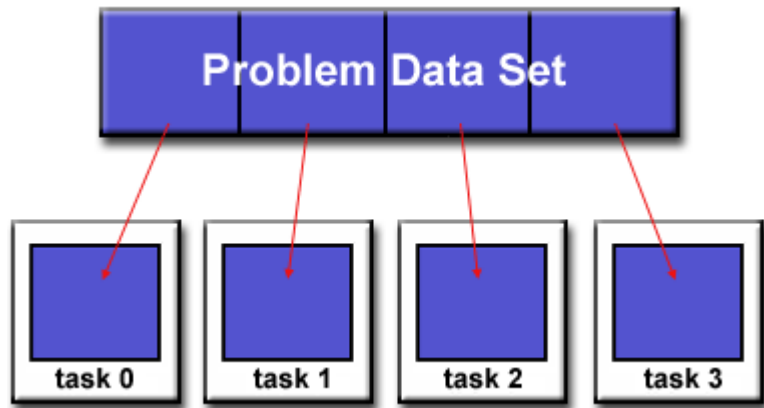
Enchantingly Parallel Code: Code without dependencies

Data Decomposition

Considerations

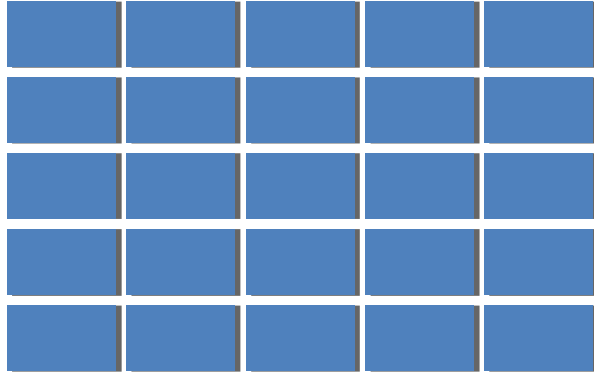
(Geometric Decomposition)

Data Structures must be (commonly) divided in arrays or logical structures.

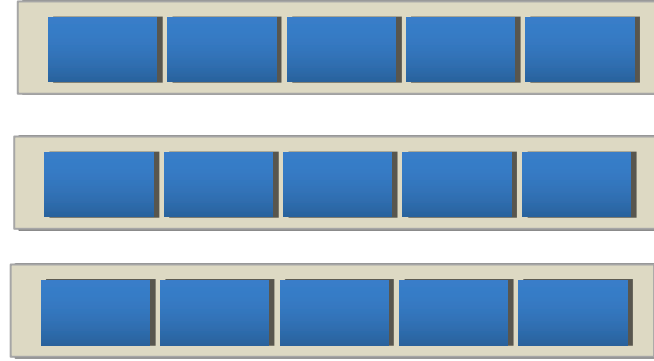


- How should you divide the data into chunks?
- How should you ensure that the tasks for each chunk have access to all data required for update?
- How are the data chunks assigned to threads?

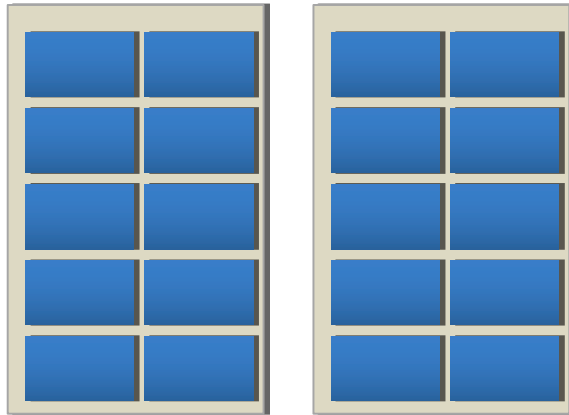
How should you divide data into chunks?



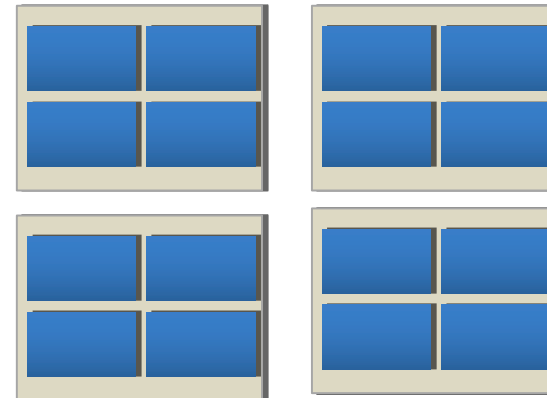
By individual elements



By rows



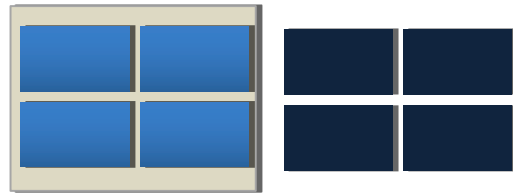
By groups of columns



By blocks

The Shape of the Chunk

- Data Decomposition have an additional dimension.
- It determines what the neighboring chunks are and how any exchange of data will be handled during the course of the chunk computations.



2 Shared Borders

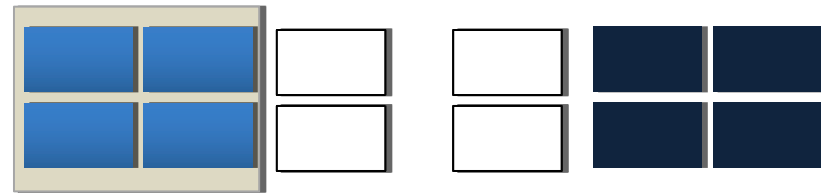


5 Shared Borders

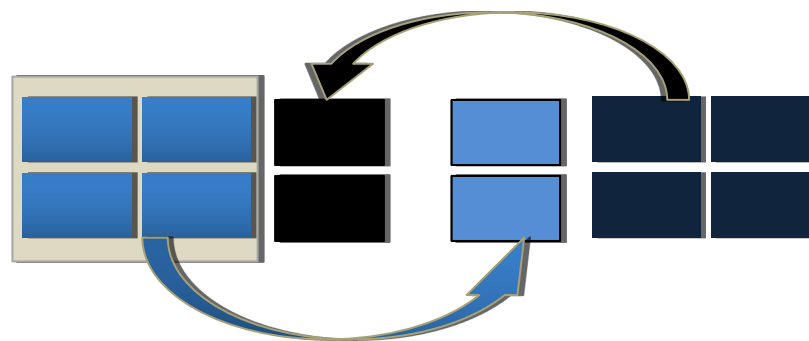
- Regular shapes : Common Regular data organizations.
- Irregular shapes: may be necessary due to the irregular organizations of the data.

How should you ensure that the tasks for each chunk have access to all data required for update?

- Using Ghost Cells
 - Using ghost cells to hold copied data from a neighboring chunk.



Original split with ghost cells



Copying data into ghost cells

Tasks and Domain Decomposition Patterns

- Task Decomposition Pattern
 - Understand the computationally intensive parts of the problem.
 - Finding Tasks (as much...)
 - Actions that are carried out to solve the problem
 - Actions are distinct and relatively independent.
- Data Decomposition Pattern
 - Data decomposition implied by tasks.
 - Finding Domains:
 - Most computationally intensive part of the problem is organized around the manipulation of large data structure.
 - Similar operators are being applied to different parts of the data structure.
 - In shared memory programming environments, data decomposition will be implied by task decomposition.

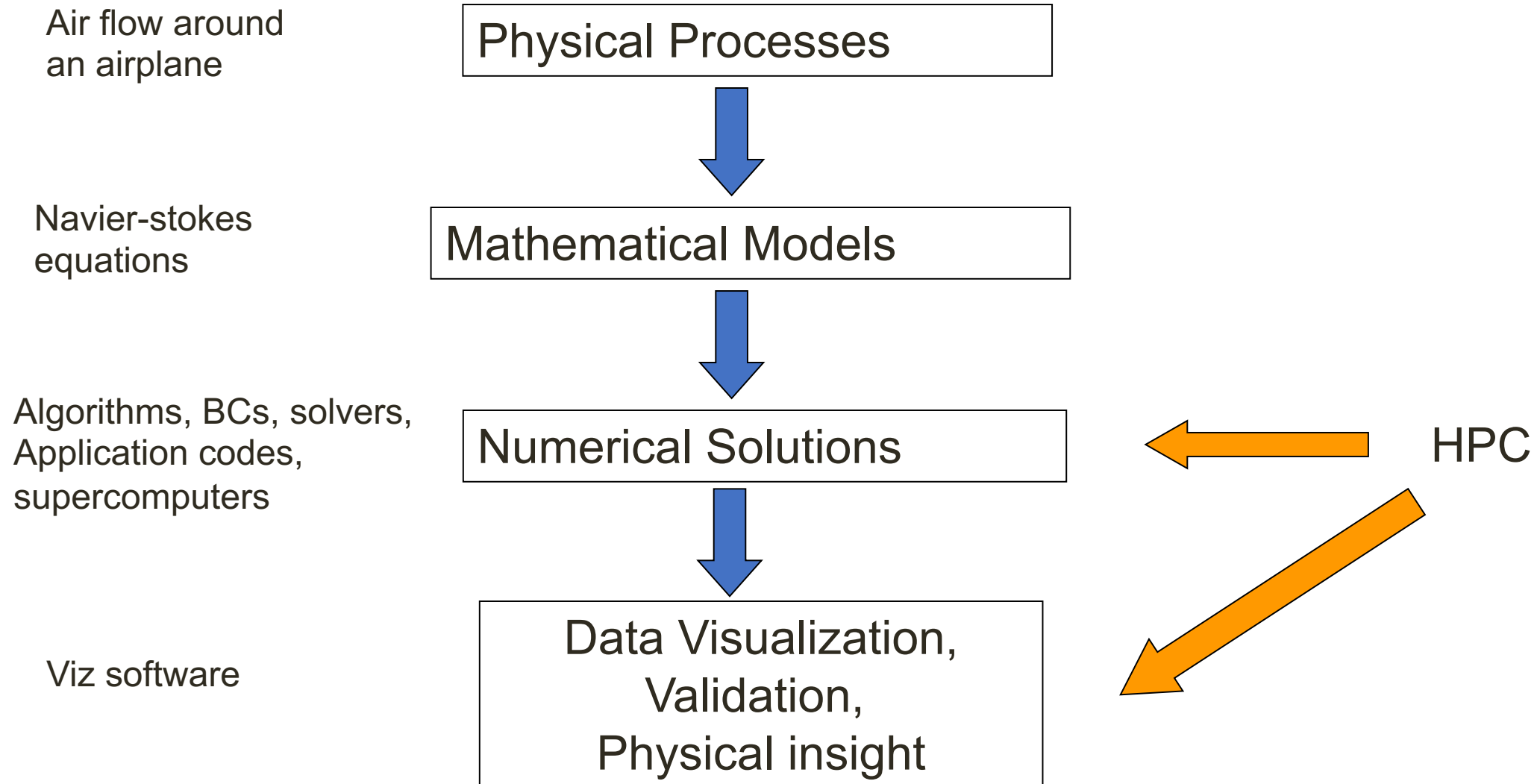
Not Parallelizable Jobs, Tasks and Algorithms

- Algorithms with state
- Recurrences
- Induction Variables
- Reductions
- Loop-carried Dependencies



The Mythical Man-Month: Essays on Software Engineering. By Fred Brooks. Ed Addison-Wesley Professional, 1995

How HPC fits into Scientific Computing



Advantages of Parallelization

- Cheaper, in terms of Price/Performance Ratio
- Faster than equivalently expensive uniprocessor machines
- Handle bigger problems
- More scalable: the performance of a particular program may be improved by execution on a large machine
- More reliable: In theory if processors fail we can simply use others

Concurrent Design Models Features

- **Efficiency**
 - Concurrent applications must run quickly and make good use of processing resources.
- **Simplicity**
 - Easier to understand, develop, debug, verify and maintain.
- **Portability**
 - In terms of threading portability.
- **Scalability**
 - It should be effective on a wide range of number of threads and cores, and sizes of data sets.

Design Evaluation Pattern

- Production of analysis and decomposition:
 - Task decomposition to identify concurrency
 - Data decomposition to indentify data local to each task
 - Group of task and order of groups to satisfy temporal constraints
 - Dependencies among tasks
- Design Evaluation
 - Suitability for the target platform
 - Design Quality
 - Preparation for the next phase of the design

Algorithm Structures



Organizing by Tasks

Task Parallelism
Divide and Conquer



Organizing by Data Decomposition

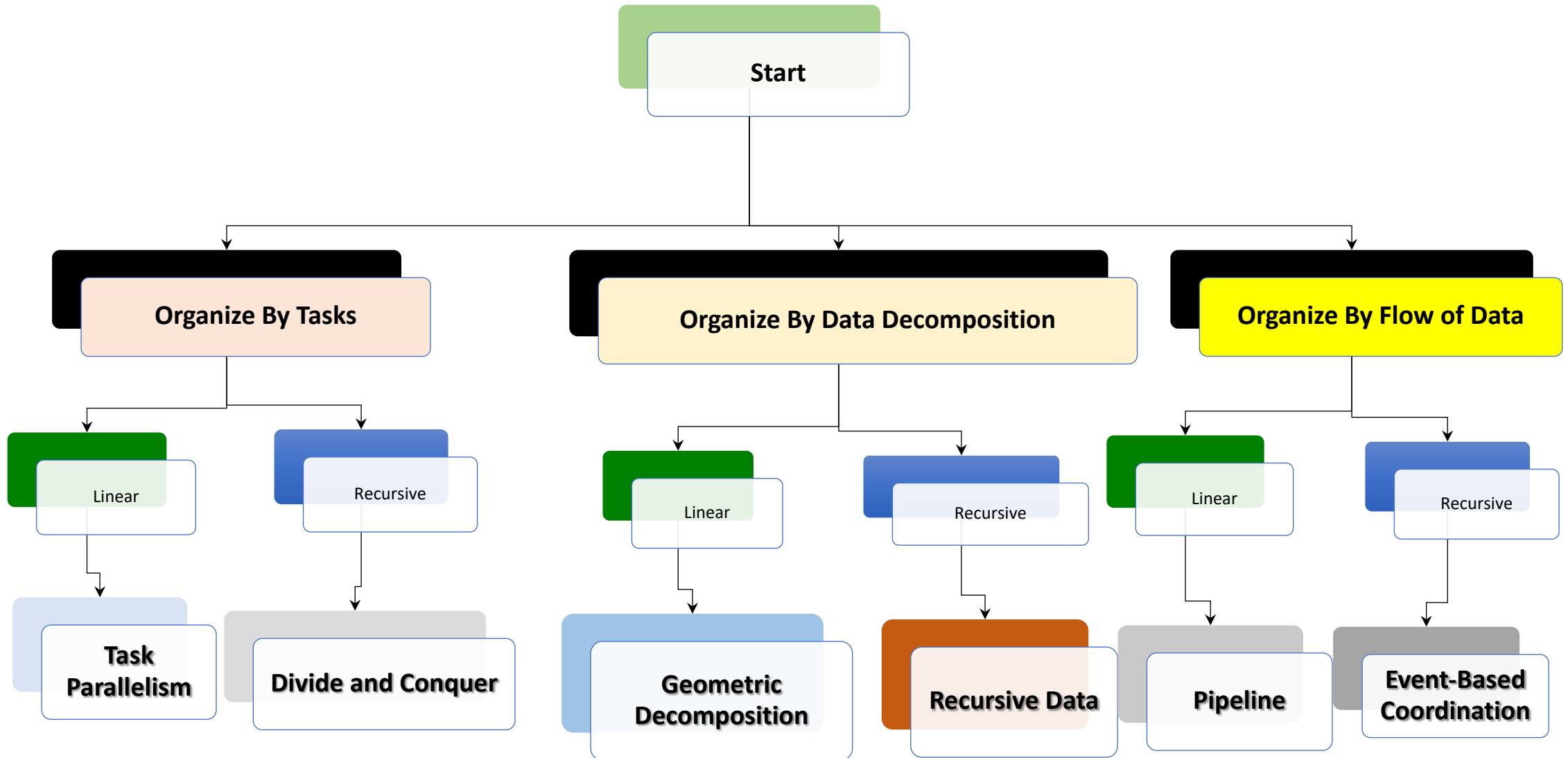
Geometric Decomposition
Recursive Data



Organizing by Flow of Data

Pipeline
Event-Based Coordination

Algorithm Structure Decision Tree (Major Organizing Principle)

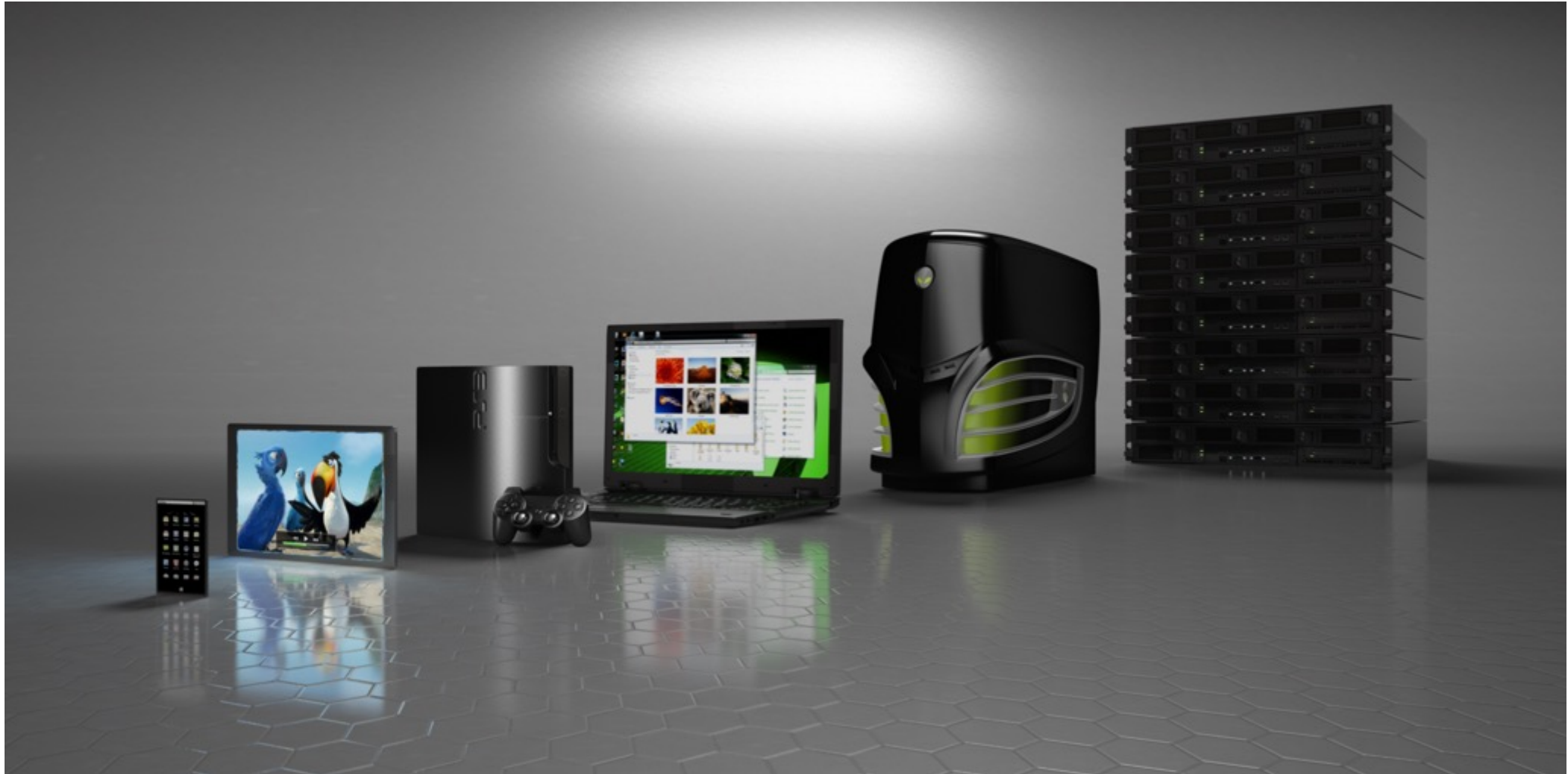


How to Exploit (Better) Concurrency

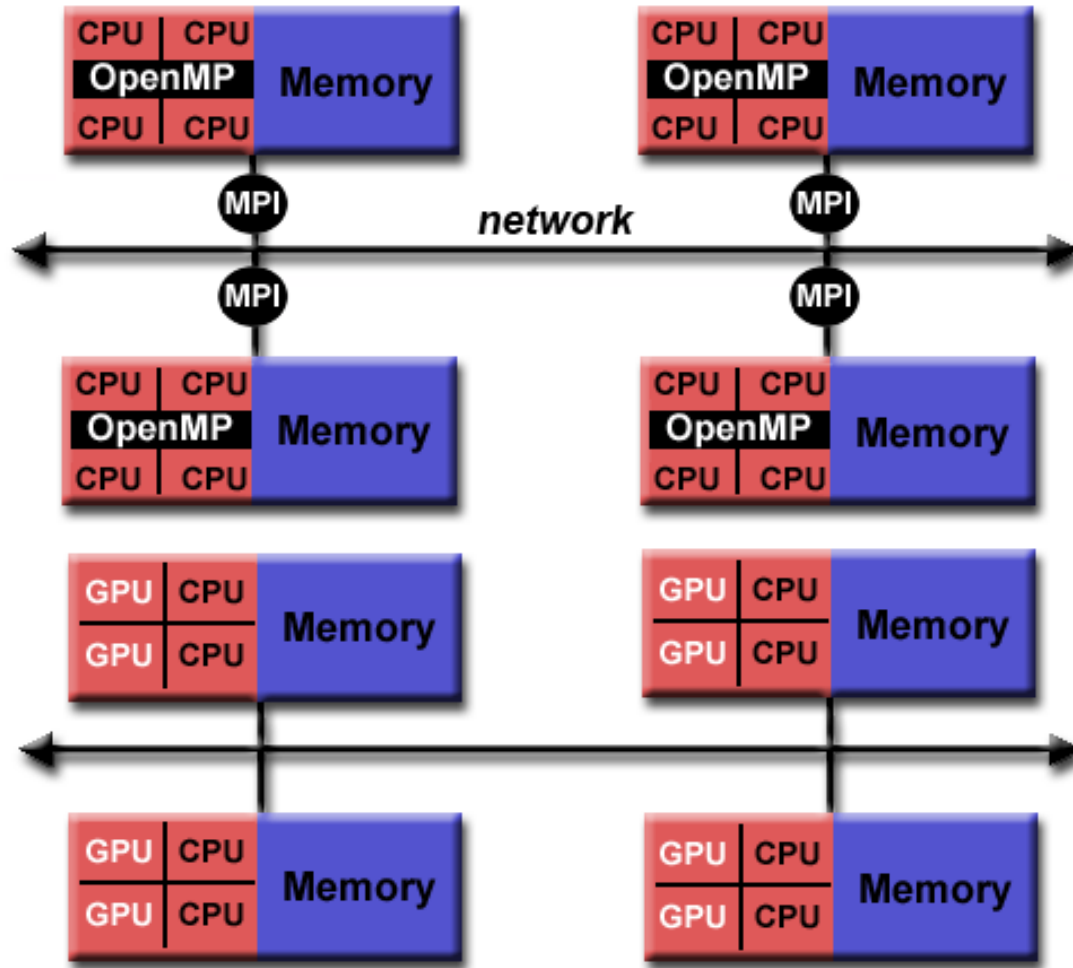
- (Remember) Mixed Approach (Algorithms/Applications - Hardware/System).
- Good Techniques from Software Engineering
- Good Problem knowledge from scientific (domain) expertise
- Confrontation and Performance Evaluation



4 Good Practice: Know your HPC support Platform

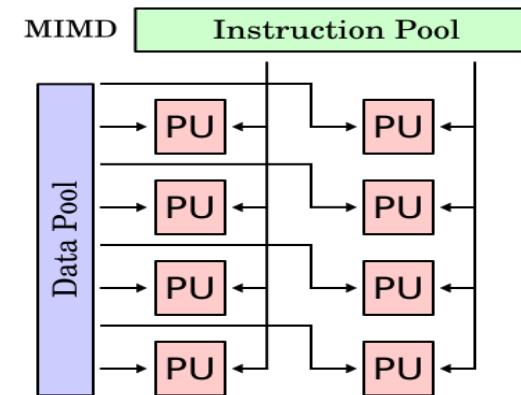
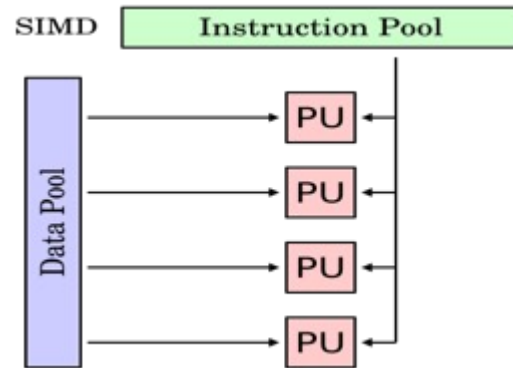
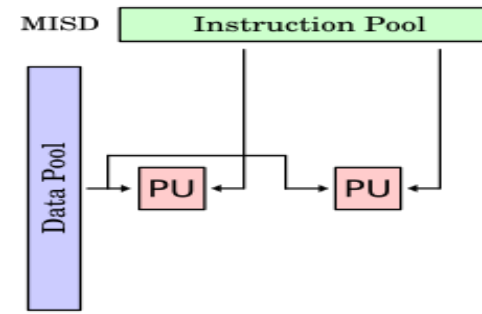
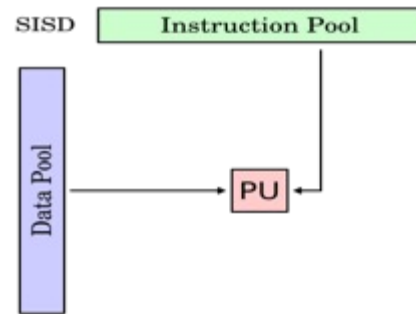


Shared, Distributed and Hybrid Memory Architectures



- Memory Exploitation involves Memory Hierarchy
 - Models as PRAM, BSP, etc..
- All modern architectures to HPC allows different memory models
 - Shared Memory (Inside Nodes)
 - Distributed Memory (Among Nodes)
 - Hybrid Memory
 - Using Accelerators (GPUs, MICs)
 - Interaction Nodes/Processors

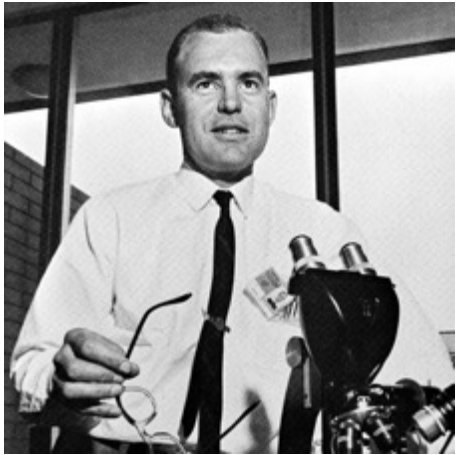
Flynn's Taxonomy*



* Proposed by M. Flynn in 1966

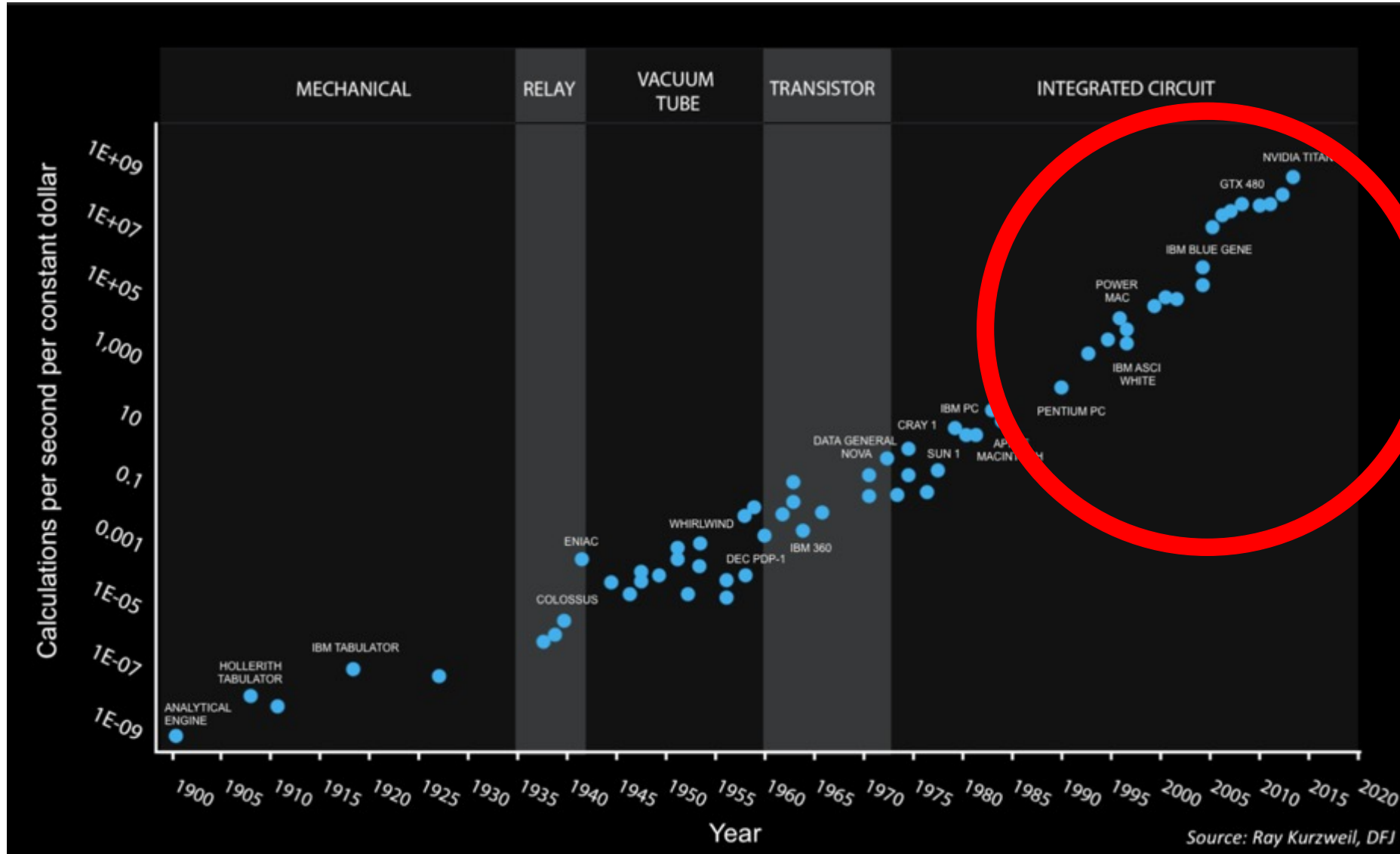
Our World
in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Gordon Moore (In the 60's)

The (Post) Moore Era



After 120 years...
The Moore's Law
is Dead

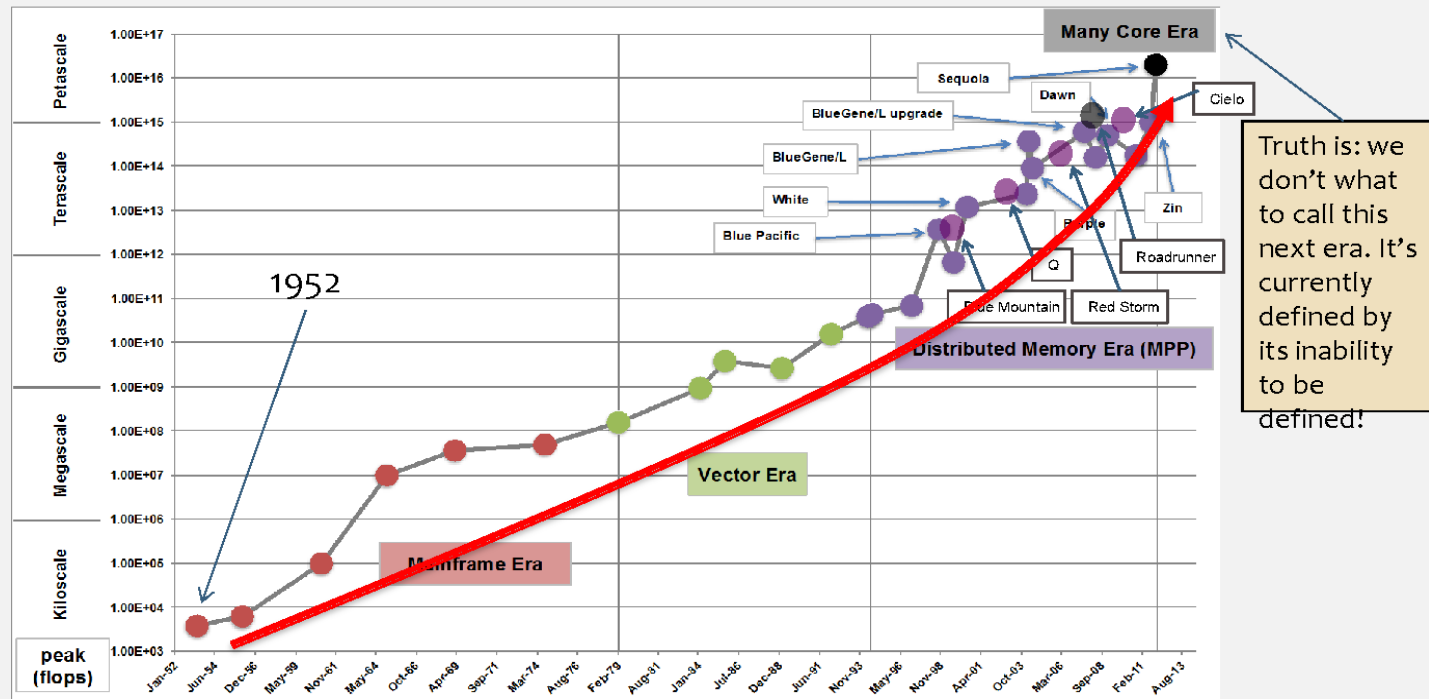


Jack Dongarra

Parallel Computing Evolution

(From the LLNL Vision by Rob Neely)

Advancements in (High Performance) Computing Have Occurred in Several Distinct “Eras”



Each of these eras define not so much a common hardware architecture, but a common programming model



Rob Neely

Configurable Architectures

Large Scale Cores
(High Single Thread
Performance)

Dual Cores

(Symmetric Multithreading)

MultiCore
Arrays

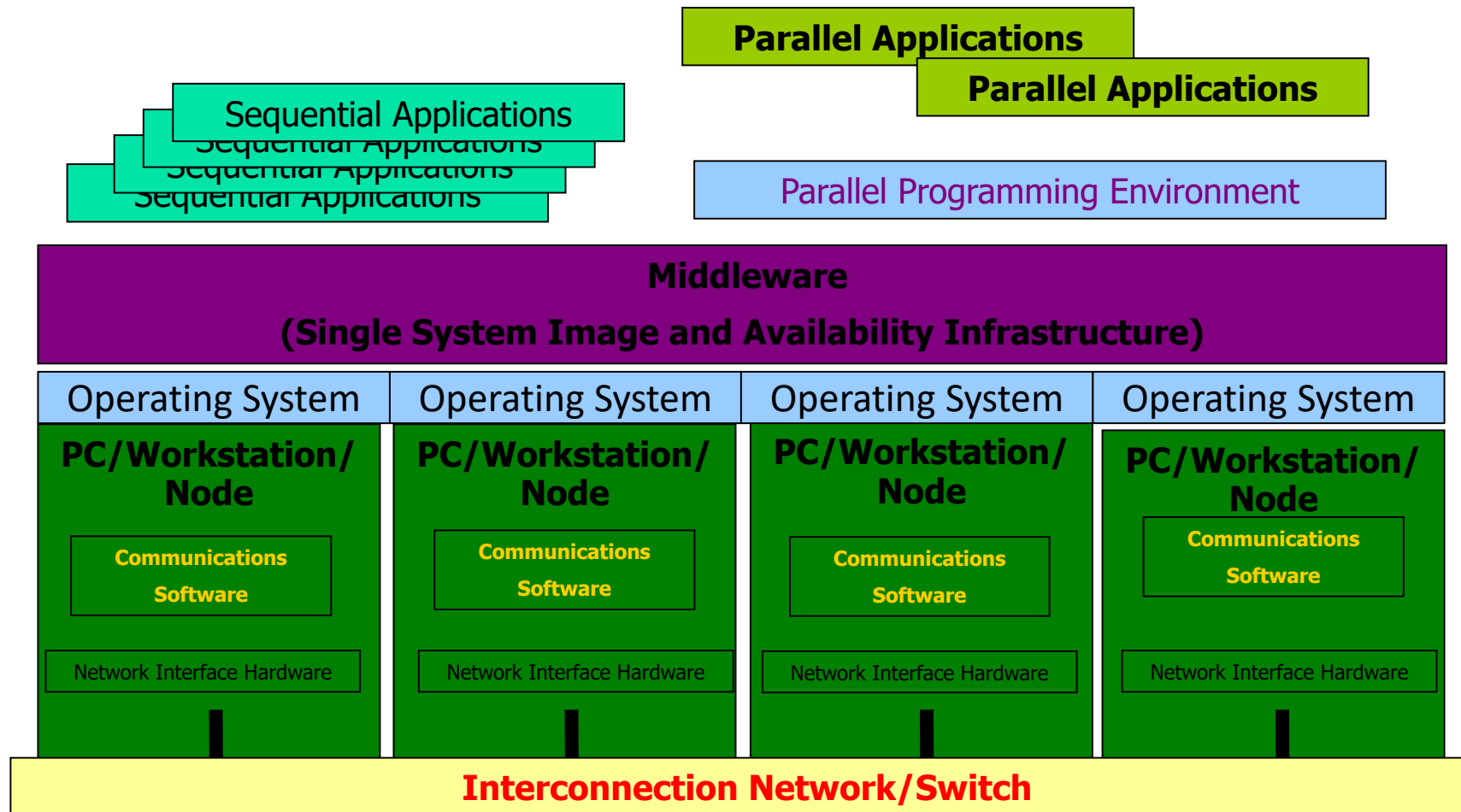
Scalar + Many
Cores

(Highly threaded
workloads)

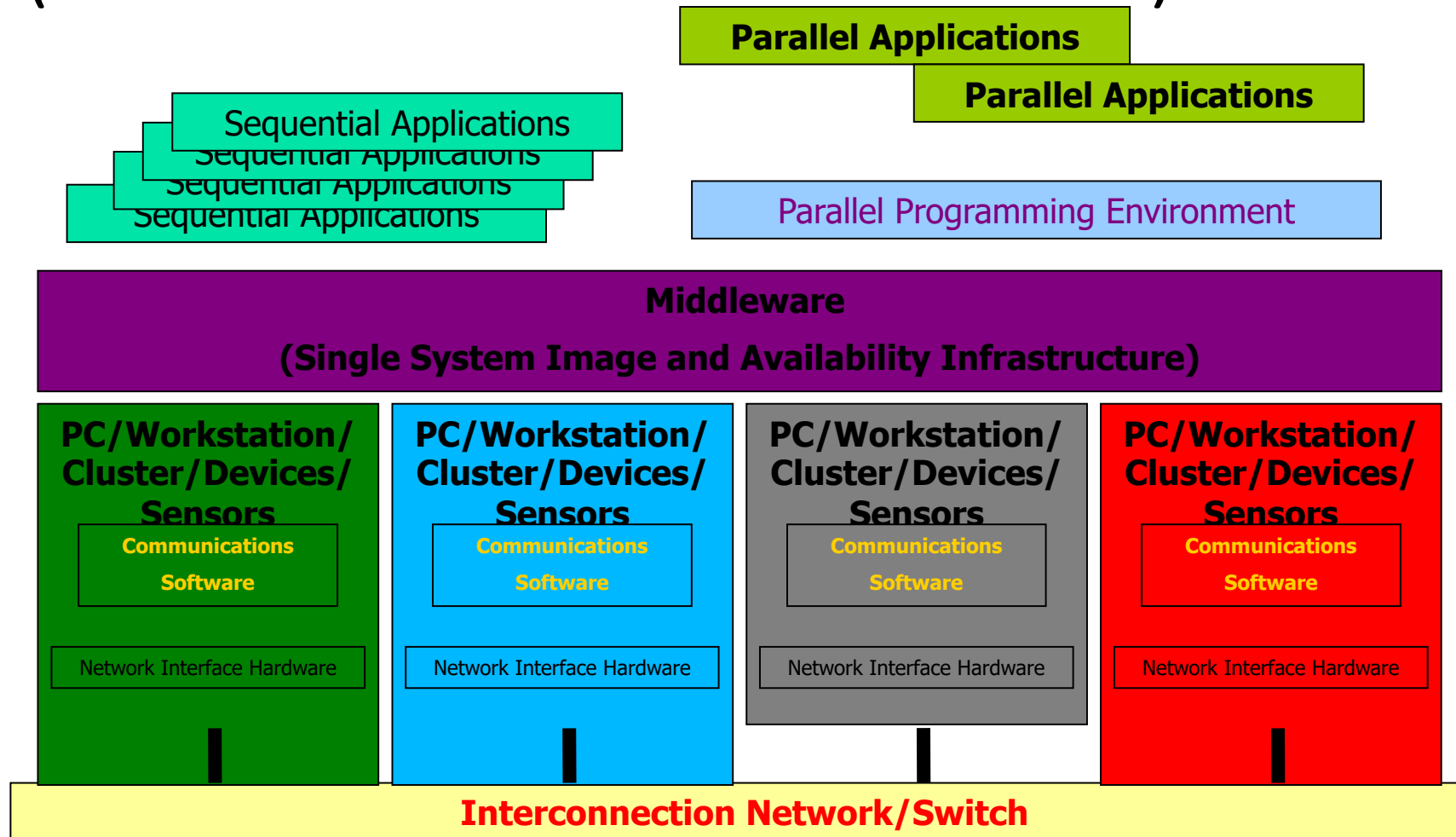
Manycore
arrays



Cluster Computing Architecture



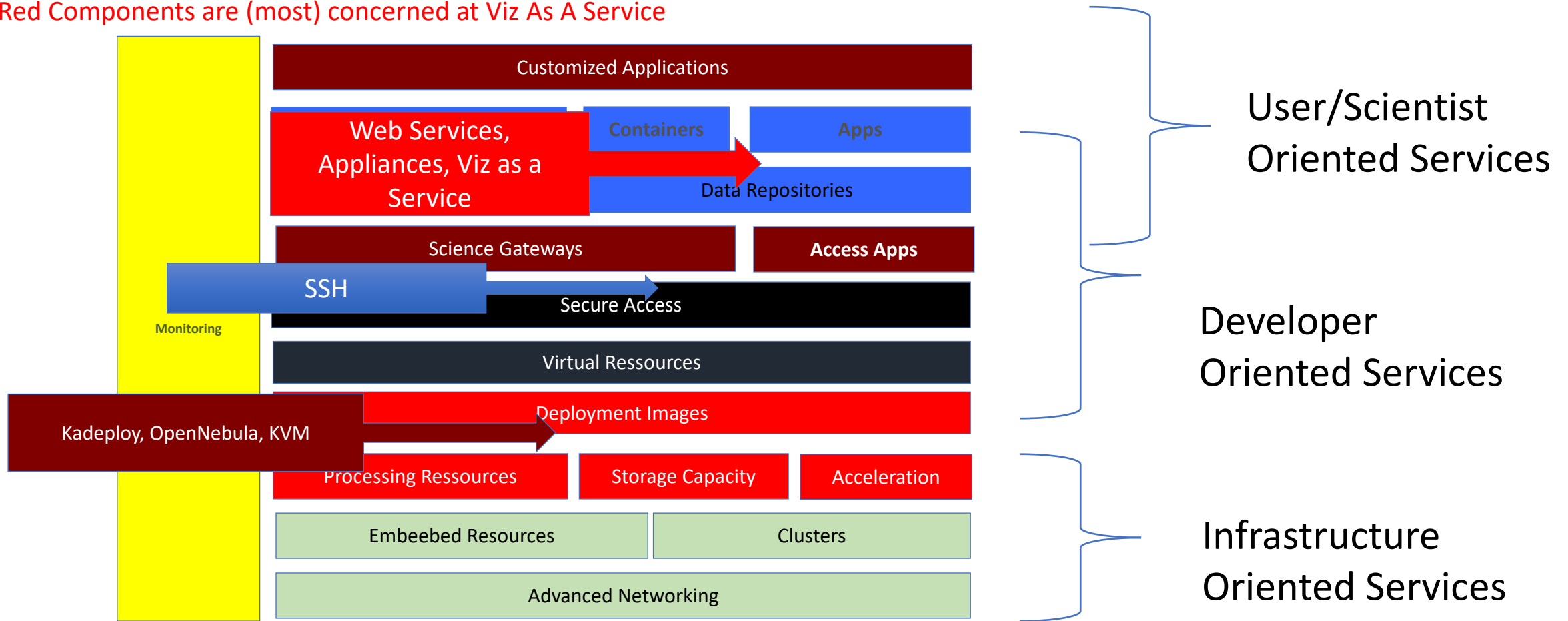
Grid Computing Architecture (Remember the Cluster Architecture)



How Exploit HPC Architectures with Cloud Visibility Models?

HPC as A Service Model

* Red Components are (most) concerned at Viz As A Service



5. Good Practice: Know Your Bugs



From

Xavier Besseron [Know Your Bugs: Weapons for Efficient Debugging](#)

Tools for debugging

Compilers

- It's the first program to check your code
- [GCC](#), [Intel Compiler](#), [CLang](#), MS Compiler, ...

Static code analyzers

- Check the program without executing it
- Splint, Cppcheck, Coccinelle, ...

Debuggers

- Inspect/modify a program during its execution
- [GDB: the GNU Project Debugger](#) for serial and multi-thread programs
- Parallel debuggers (commercial): RogueWave Totalview, Allinea DDT

Dynamics code analyzers and profilers

- Check the program while executing it
- [Valgrind](#), Gcov, Gprof, CLang sanitizers, ...
- Commercial software: Purify, Intel Parallel Inspector, ..

Compilers

What does a compiler do?

- Translate source code to machine code
 - 3 phases:
 - Lexical analysis: recognize "words" or tokens
 - Syntax analysis: build syntax tree according to language grammar
 - Semantic analysis: check rules of the language, variable declaration, types, etc.
 - With this knowledge, a compiler can find many bugs
- Pay attention to compiler warnings and errors of a program

A compiler can find out if your program makes sense according to the language. However, it cannot guess what you are trying to do.

How to use the compiler

- Choose your compiler

	GCC	CLang	Intel Compiler
C	gcc	clang	icc
C++	g++	clang++	icpc
Fortran	gfortran		ifort

- Activate warning messages with the `-Wall` parameters
- Warnings can be enabled/disabled individually, *cf* documentation
- Compile with debug symbols with `-g` parameters

- Example

- ```
$ gcc -g -Wall program.c -o program
```

```
program.c: In function 'main':
```

- ```
program.c:4:15: error: 'y' undeclared (first use in this function)
```

- ```
int z = x + y;
```

- ```
^
```

- ```
program.c:4:15: note: each undeclared identifier is reported only once for each
```

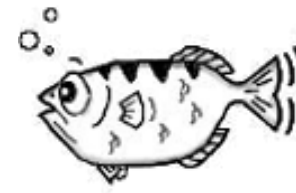
```
program.c:4:7: warning: unused variable 'z' [-Wunused-variable]
```

- ```
int z = x + y;
```

- ```
^
```



# GNU Debugger 1/2



## GDB is the GNU Debugger

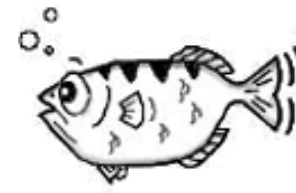
- Allow to execute a program step by step
- Watch the value of variables
- Stop the execution on given condition
- Show the backtrace of an error
- Modify value of variables at runtime

## Starting GDB

- Compile your program with the `-g` option
- Start program execution with GDB  
`gdb --args myprogram arg1 arg2`
- Or open a core file (generated after a crash)  
`gdb myprogram corefile`

# GNU Debugger 2/2

## Using GDB



- Command line tool
- Many graphical frontends available too: [DDD](#), [Qt Creator](#), ...
- Online documentation & tutorial:

<http://sourceware.org/gdb/current/onlinedocs/gdb/>

[http://www.cs.swarthmore.edu/~newhall/unixhelp/howto\\_gdb.html](http://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.html)

## Main commands

- `help <command>`: get help about a command
- `run`: start execution
- `continue`: resume execute
- `next`: execute the next line
- `break`: set a breakpoint at a given line or function
- `bracktrace`: show the backtrace
- `print`: print the value of a variable
- `quit`: quit GDB

# Logic and syntax bugs

## Due to careless programming

- Infinite loop / recursion
- Confusing syntax error,  
eg use of = (affectation) instead of == (equality)
- Hard to detect, <sup>87/100</sup> because everything is correct in your mind

## What to do?

- Compile with warnings enabled
- Get some rest and/or an external advice

# Integer overflow 1/2

## Integer variables have limited size

|                                     | Size    | Minimum   | Maximum      |
|-------------------------------------|---------|-----------|--------------|
| signed short                        | 16 bits | $-2^{15}$ | $2^{15} - 1$ |
| unsigned short                      | 16 bits | 0         | $2^{16} - 1$ |
| signed int                          | 32 bits | $-2^{31}$ | $2^{31} - 1$ |
| <small>88 / 38</small> unsigned int | 32 bits | 0         | $2^{32} - 1$ |
| signed long long int                | 64 bits | $-2^{63}$ | $2^{63} - 1$ |
| unsigned long long int              | 64 bits | 0         | $2^{64} - 1$ |

If the result of an operation cannot fit in the variable,  
most-significant bits are discarded  
⇒ we have an **Integer Overflow**

# Integer overflow 2/2

## Overflow example

```
unsigned char A = 200;
```

```
unsigned char B = 60;
```

```
//Overflow!
```

```
unsigned char S = A + B;
```

|       |   |   |   |   |   |   |   |   |   |      |
|-------|---|---|---|---|---|---|---|---|---|------|
|       | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |   | 200  |
| +     | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |   | + 60 |
| <hr/> |   |   |   |   |   |   |   |   |   |      |
| =     | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | = 4  |

→ No error at runtime!

## What to do?

- Use the right integer type for your data
- In C/C++/Fortran, overflow needs to be checked manually
- CLang and GCC 5.X offer builtin functions to check for overflow  
\_\_builtin\_add\_overflow, \_\_builtin\_sub\_overflow,  
\_\_builtin\_mul\_overflow, ...

# Floating-Point Number bugs 1/2

## Floating-Point Exceptions (FPE)

- Division by zero:

$$\frac{X}{0.0} = \infty$$

- Invalid operation:

$$\sqrt{-1.0} = \text{NaN (Not A Number)}$$

- Overflow / Underflow:

$$e^{1e30} = \infty \qquad e^{-1e30} = 0.0$$

## Loss of precision

- The order of the operations matters:

$$(10^{60} + 1.0) - 10^{60} = 0.0$$

$$(10^{60} - 10^{60}) + 1.0 = 1.0$$

# Floating-Point Number bugs 2/2

## Floating-Point Exceptions and Errors

- No error at runtime by default
- Errors can propagate through all the computation

## What to do?

- Enable errors at runtime in C/C++

```
91 / 38
#define_GNU_SOURCE
#include<fenv.h>

int main()
{
 feenableexcept (FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW) ;
 ...
}
```

- Read "*What Every Computer Scientist Should Know About Floating-Point Arithmetic*" by David Goldberg

# Memory allocation/deallocation

## Dynamic memory management in C

- `void *p = malloc(size)` allocates memory
- `free(p)` de-allocates the corresponding memory
- In C++, equivalents are `new` and `delete` operations

## Common mistakes

- Failed memory allocation
- Free non-allocated memory
- Free memory twice (double free error)

These mistakes might not trigger an error immediately  
Later on, they can cause **crashes** and **undefined behavior**

## What to do?

- Check return code (cf documentation)
- Use **Valgrind** with `--leak-check=full` to catch it



# Memory leaks

## Memory is allocated but never freed

- Allocated memory keeps growing until it fills the computer memory
- Can causes a crash of the program or of the full computer
- Very common in C program, almost impossible in Fortran, Java

93 / 38

## What to do?

- For each `malloc()`, there should be a corresponding `free()`
- Use `Valgrind` with `--leak-check=full` to catch it

# Stack overflow

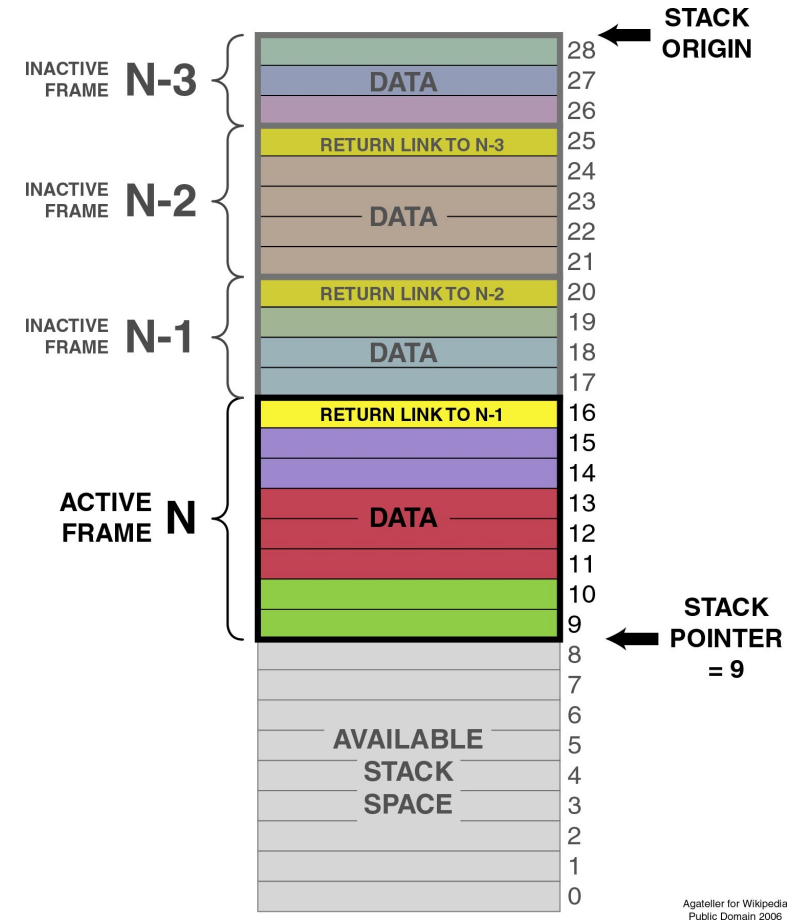
## Program stack

- Each function call create a new frame
- Function parameters and local variables are allocated in the frame

94 / 38

## Stack overflow

- Too many function calls usually not-ending recursive calls
- Oversized local data



# Buffer overflow

## Buffer overflow

- Write data in a buffer with an insufficient size
- Overwrite other data (variable, function return address)
- Can be a major security issue
- Can make the stack trace unreadable

## What to do?

- Use functions that check the buffer size:  
`strcpy()` → `strncpy()`, `sprintf()` → `snprintf()`, etc.
- GCC option `-fstack-protector` checks buffer overflow

# Out of bound access

## Read/write outside of the bound of an array

- Mismatch in the bound of an array:  $[0, N - 1]$  in C,  $[1, N]$  in Fortran
- Out of bound reading can cause undefined behavior
- Out of bound writing can cause memory corruption

96 / 38

## What to do?

- Use [Valgrind](#), it should show error  
Invalid read/write of size X

# Input/Output errors

## Errors when reading/writing in files

- Usually have an external cause:
  - Disk full
  - Quota exceeded
  - Network interruption
- System call will 97 / 38 return an error or hang

## What to do?

- Always can check the return code
- Usually stop execution with an explicit message

# Race condition Bugs

"Debugging programs containing race conditions is no fun at all."  
Andrew S. Tanenbaum, *Modern Operating Systems*

## Race condition

98 / 38

- A timing dependent error involving shared state
- It runs fine most of the time, and from time to time, something weird and unexplained appears

## Different kind of race conditions

- **Data race**: Concurrent accesses to a shared variable
- **Atomicity bugs**: Code does not enforce the atomicity for a group of memory accesses, eg Time of check to time of use
- **Order bugs**: Operations are not executed in order  
Compilers and processors can actually re-order instructions

## What to do?

- Protect critical sections: **Mutexes**, **Semaphores**, etc.
- Use atomic instructions and memory barriers (low level)
- Use compiler builtin for atomic operations<sup>2</sup> (higher level)

---

<sup>2</sup>[https://gcc.gnu.org/onlinedocs/gcc-5.1.0/gcc/\\_005f\\_005fatomic-Builtins.html](https://gcc.gnu.org/onlinedocs/gcc-5.1.0/gcc/_005f_005fatomic-Builtins.html)

# Deadlock 1/3



[Deadlock](#), photograph by David Maitland

## What to do?

- Think before writing multithread code
- Use high level programming model: [Open MP](#), [Intel TBB](#), [MPI](#), etc.
- Theoretical analysis
- Software for thread safety analysis

*"I would love to have seen them go their separate ways, but I was exhausted. The frog was all the time trying to pull the snake off, but the snake just wouldn't let go."*



# Performance bugs

## Bad Performance can be seen as a bug

- Bad algorithm: too high computation complexity  
Example: *Insertion Sort* is  $O(N^2)$ , *Quick Sort* is  $O(N.\log(N))$
- Memory copies can be a problem,  
specially with Object Oriented languages
- Some memory allocator have issues:  
memory alignment constraints, multithread context

## What to do?

- Try use existing proven libraries when possible:  
eg Eigen library for linear algebra, C++ STL, Boost, etc.
- Use a profiler to see where your program spend most of its time  
[Valgrind](#) with [Callgrind](#), [GNU gprof](#), many commercial tools ...
- ...

## 4 Good practices to catch bugs



# Be a good programmer

## Write good code

- Use explicit variable names, don't re-use variable
- Avoid global variables (problematic in multi-threads)
- Comment and document your code
- Keep your code simple, don't try to over-optimize

## Use defensive programming

- Add assertions, 103 / 38 *cf* `assert()`
- Always check return codes, *cf* manpages and documentation

## Re-use existing libraries

- Use existing libraries when available/possible
  - Probably better optimized and tested than your code
- ⇒ Code easier to understand and maintain
- ⇒ Catch bugs as soon as possible

# Compilers and Tests

## Use your compilers

- Enable (all) warnings of the compiler
- Vary the compilers and configurations
  - Different compilers (GCC, CLang, Intel Compiler, MS Compiler)
  - Various architectures (Windows/Linux, x86/x86\_64/ARM)

## Testing and Code Checking

- Write unit tests and regression tests
  - Use coverage analysis tools
  - Use static and dynamic code analysis tools
  - Continuous integration:
    - Frequent compilation, testing, execution
    - Different configurations and platforms
- ⇒ Catch more warnings and errors
- ⇒ Better portability

# Know your tools

## Know the error messages

- Look in the documentation / online
- Compiler errors/warnings
- Runtime errors:  
Segmentation fault, Floating point exception, Double free, etc.
- Valgrind errors:  
Invalid read of size 4  
Conditional jump or move depends on uninitialised value(s)  
8 bytes in 1 blocks are definitely lost  
...

## Use the right tool

- Know your tools and when to use them
  - GDB: locate a crash
  - Valgrind: memory-related issue
  - ...

# Debug with methodology

## Find a minimal case to reproduce the bug

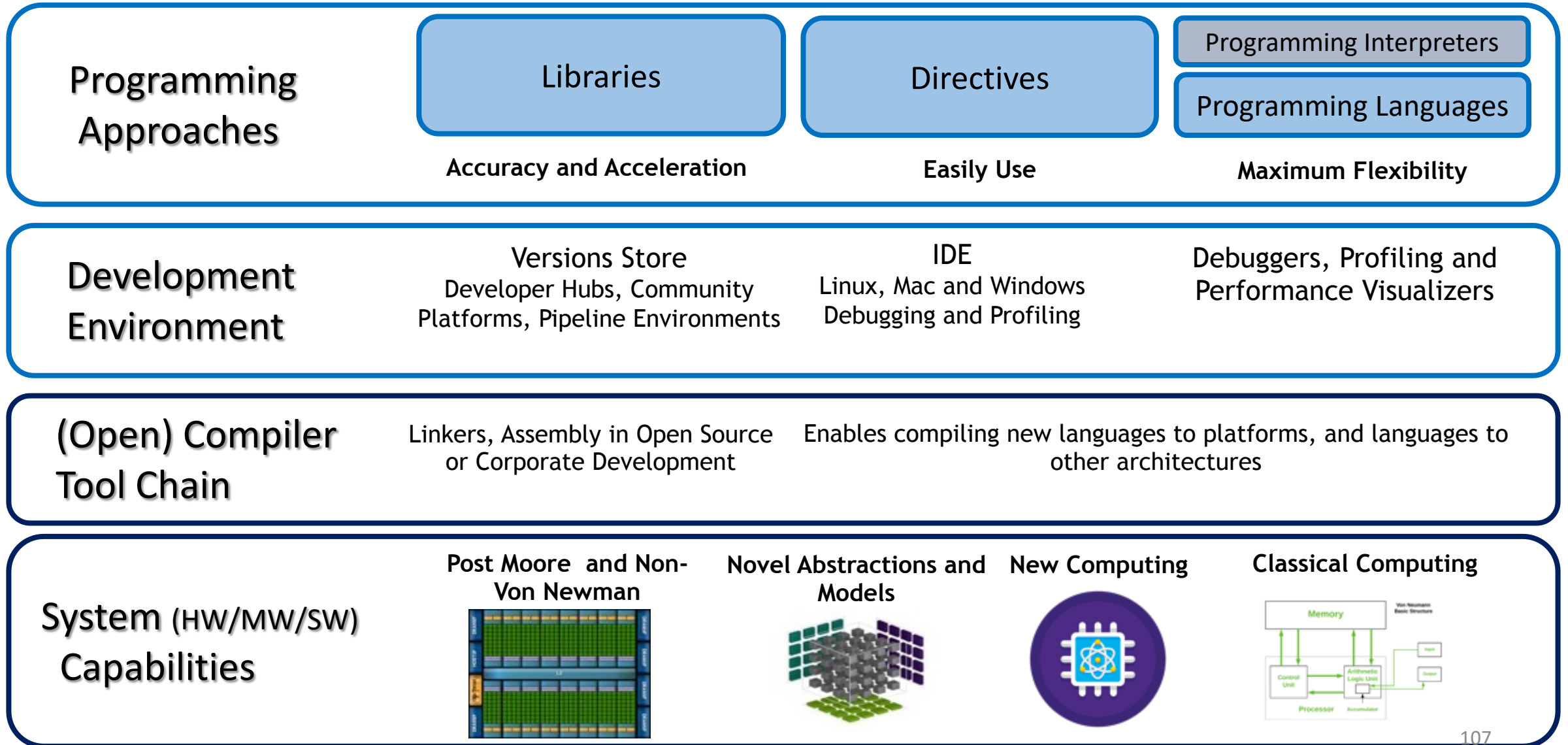
- Some bugs are intermittent
- Easier to debug
- Help you to understand the cause
- Allow to check that the bug is really fixed
- Bonus: make a regression test

## Use a Control Version System (GIT, SVN, ...)

- Keep history, serve as a backup, allow to go back in time
- GIT has a nice feature of code bisection in history to find when a bug has been introduced

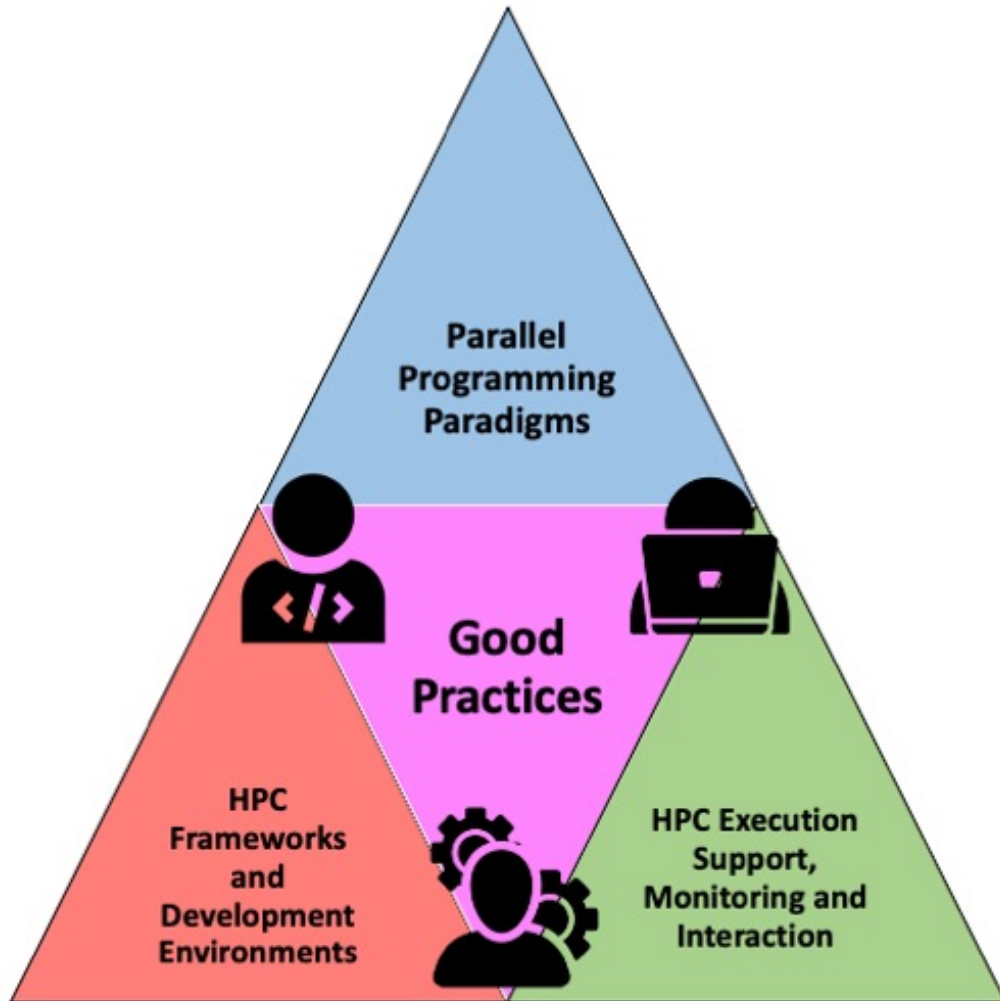
# 6. Good Practice: Known the Ecosystem

(Inspired by the Accelerated/Hybrid Computing World)



# 6. Good Practice: Known the Ecosystem

(Inspired by the Accelerated/Hybrid Computing World)



- Administration People
- Support People
- Developers
- Coders
- Users



# Important References

- C++ Programming Tutorial and Instructions for Practical Sessions by Christopher Lester  
Department of Physics (based on earlier versions by David MacKay, Roberto Cipolla and Tim Love) <https://www.hep.phy.cam.ac.uk> or a google search.
- Bjarne Stroustrup's site <https://www.stroustrup.com/>
- The C++ Foundation's site <https://isocpp.org/>
- Code Block's site: <http://www.codeblocks.org/>
- The cplusplus.com site: <https://www.cplusplus.com/>
- The C++ Point Tutorial's Site <https://www.tutorialspoint.com/cplusplus/index.htm>
- Learn C++ <https://www.learn-cpp.org/>

# Information Resources:

- An Introduction to the C Programming Language and Software Design by Tim Bailey (Free document)
- C Language Tutorial (Free document)
- **C Programming Learn to Code Sisir Kumar Jena (Non free, however requested in the campus : <https://www.routledge.com/C-Programming-Learn-to-Code/Jena/p/book/9781032036250> )**
- TENOUK'S C & C++ RANT (Available in: <https://www.tenouk.com/> and <https://www.tenouk.com/clabworksheet/clabworksheet.html> )

# Recommended Lectures

- **The Art of Concurrency “A thread Monkey’s Guide to Writing Parallel Applications”**, by *Clay Breshears* (Ed. O Reilly, 2009)
- **Writing Concurrent Systems. Part 1.**, by *David Chisnall* (InformIT Author’s Blog: <http://www.informit.com/articles/article.aspx?p=1626979> )
- **Patterns for Parallel Programming.**, by T. Mattson., B. Sanders and B. MassinGill (Ed. Addison Wesley, 2009) Web Site: <http://www.cise.ufl.edu/research/ParallelPatterns/>
- Designing and Building Parallel Programs, by Ian Foster in <http://www.mcs.anl.gov/~itf/dbpp/>
- Lectures in the site: [www.sc-camp.org](http://www.sc-camp.org)

Thank you!

