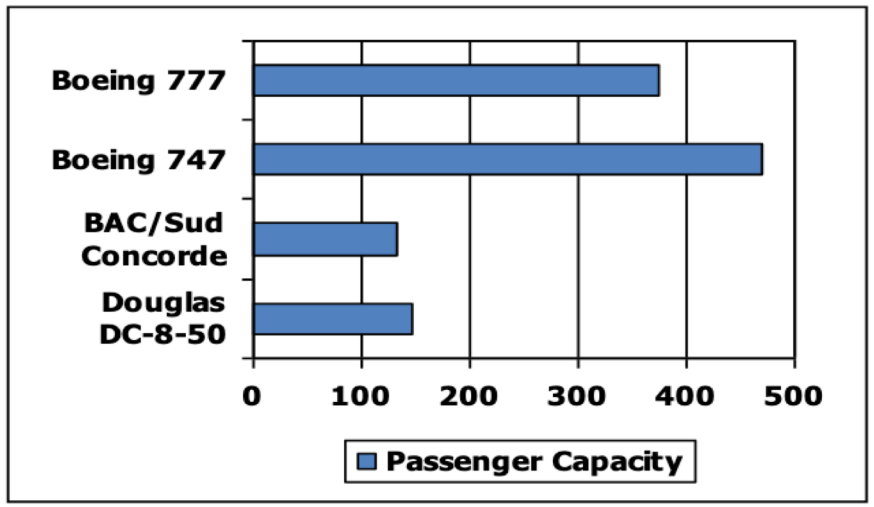
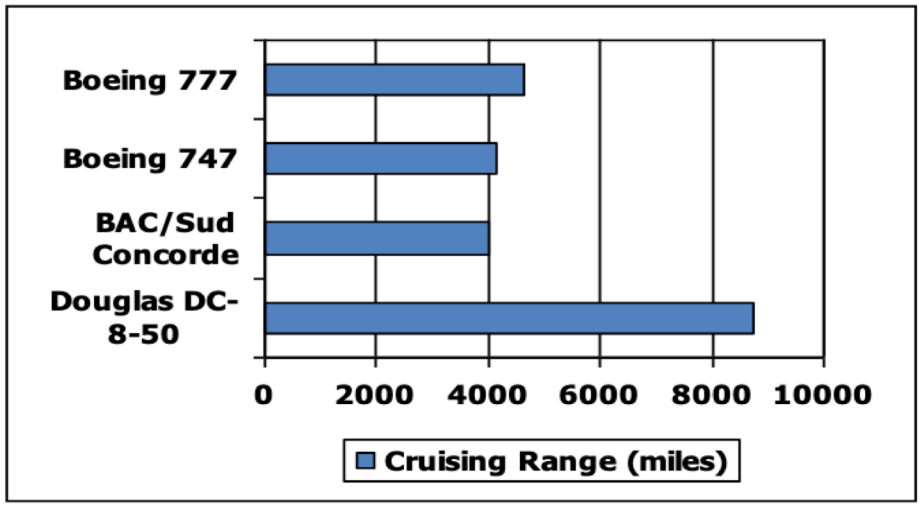
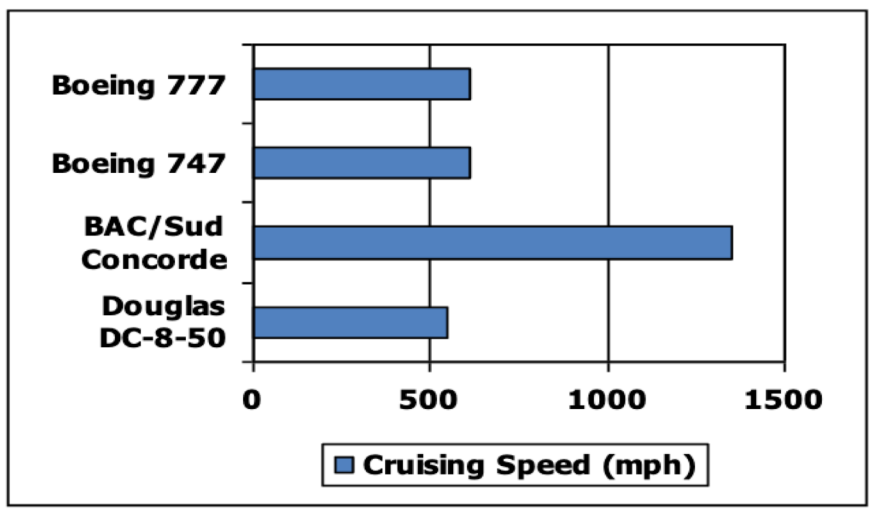
Speedup and Performance

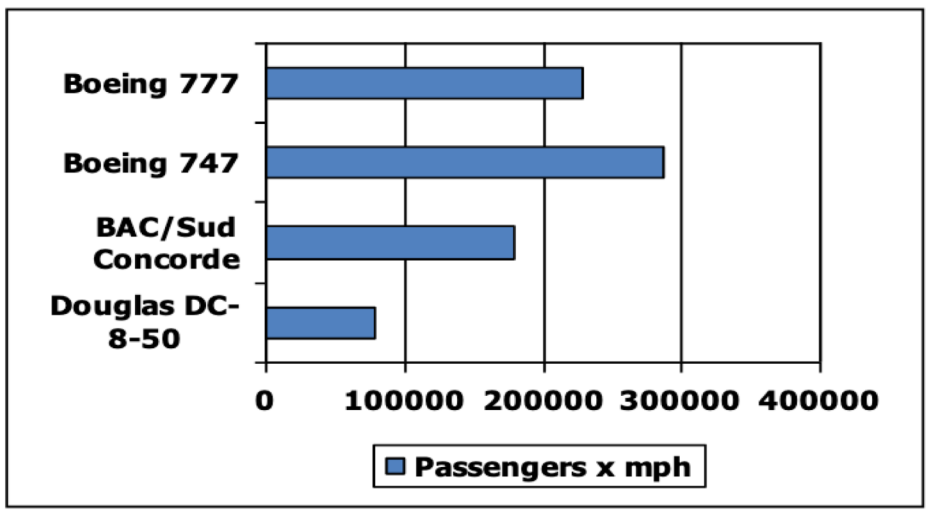
Defining Performance

- Which airplane has the best performance?









Standard Definition of Performance

- For some program running on machine X,

$$\text{Performance}_X = 1 / \text{Execution time}_X$$

- "X is n times faster than Y"

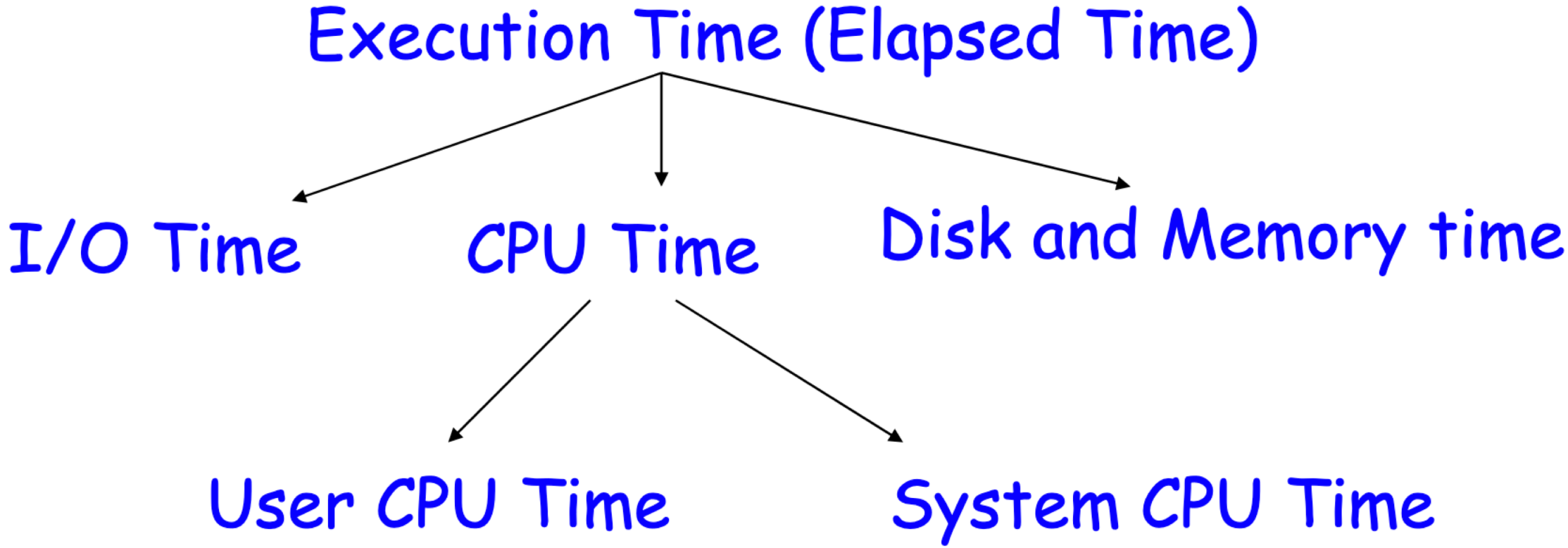
$$\text{Performance}_X \ / \ \text{Performance}_Y = n$$

  - Example: time taken to run a program
    - 10s on A, 15s on B
    - Execution Time_B / Execution Time_A = 15s / 10s = 1.5
    - So A is 1.5 times faster than B

Execution Time

- **Elapsed Time**
  - counts everything *(disk and memory accesses, I/O , etc.)*
  - a useful number, but often not good for comparison purposes
- **CPU time**
  - doesn't count I/O or time spent running other programs
  - can be broken up into system time, and user time

- Our focus: **user CPU time**
  - time spent executing the lines of code that are "in" our program


Execution Time (Elapsed Time)
I/O Time
CPU Time
Disk and Memory time
User CPU Time
System CPU Time

Basic measuring metrics

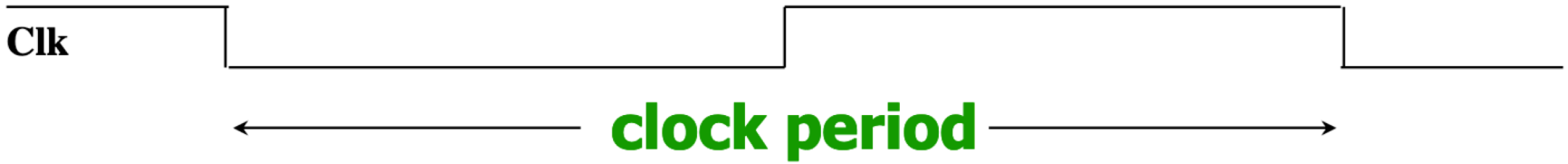
- Comparing Machines
  - Metrics
    - Execution time
    - Throughput
    - CPU time
    - MIPS – millions of instructions per second
    - MFLOPS – millions of floating point operations per second
- Comparing Machines Using Sets of Programs
  - Arithmetic mean, weighted arithmetic mean
  - Benchmarks

Let's Look at Two Simple Metrics

- Response time (aka Execution Time)
  - The time between the start and completion of a task
- Throughput
  - Total amount of work done in a given time

Computer Clock

- A computer clock runs at a constant rate and determines when events take placed in hardware.



- The clock cycle time is the amount of time for one **clock period** to elapse (e.g. 5 ns).
- The clock rate is the inverse of the clock cycle time.
- For example, if a computer has a clock cycle time of 5 ns, the clock rate is:

$$\frac{1}{5 \times 10^{-9} \text{ sec}} = 200 \text{ MHz}$$

Understanding Cycles

- A given program will require
  - some number of instructions (machine instructions)
  - some number of clock cycles
  - some number of seconds
- We have a vocabulary that relates these quantities:
  - clock cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - CPI (cycles per instruction)
    - *a floating point intensive application might have a higher CPI*

Computing CPU Time

- The time to execute a given program can be computed as

  CPU time = CPU clock cycles x clock cycle time

- Since clock cycle time and clock rate are reciprocals

  CPU time = CPU clock cycles / clock rate

- The number of CPU clock cycles can be determined by

  CPU clock cycles = (instructions/program) x (clock cycles/instruction)

  = Instruction count x CPI

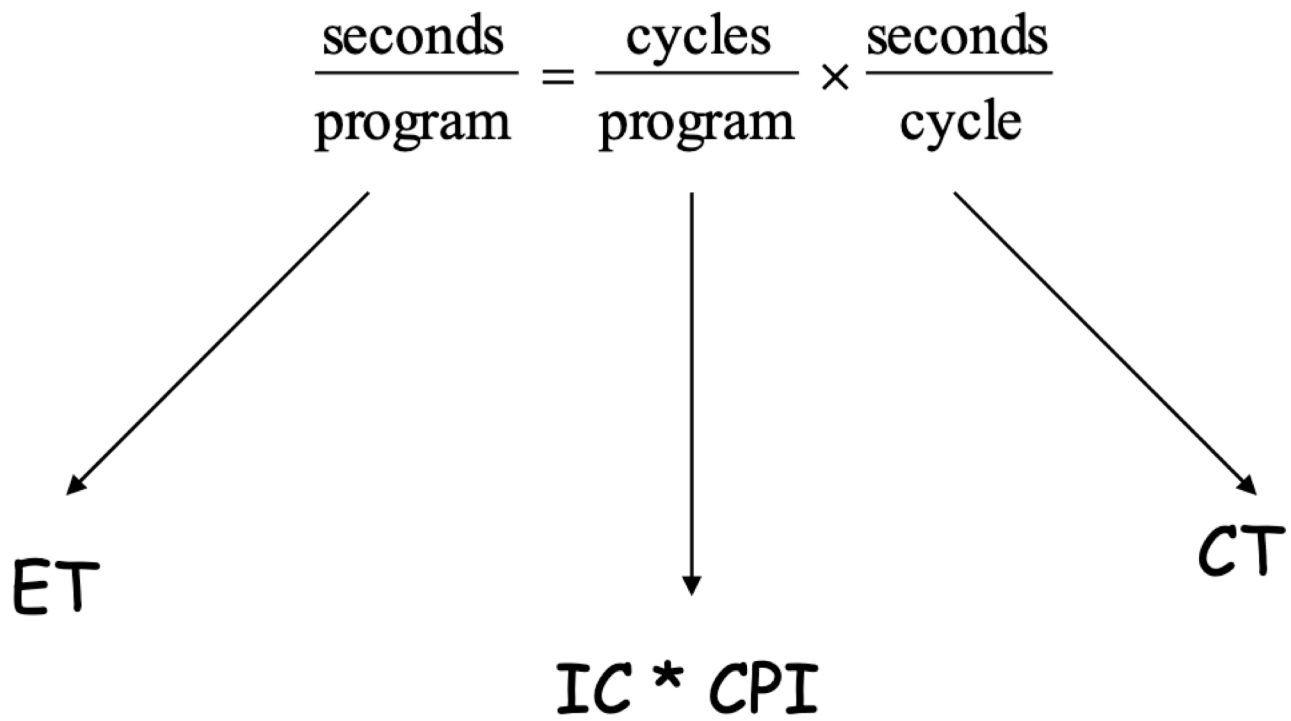
  which gives

  CPU time = Instruction count x CPI x clock cycle time

  CPU time = Instruction count x CPI / clock rate

- The units for CPU time are

$$\text{CPU time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$



ET = IC X CPI X CT

ET = Execution Time
CPI = Cycles Per Instruction
IC = Instruction Count

Performance

- Performance is determined by execution time
- Do any of the other variables equal performance?
  - # of cycles to execute program?
  - # of instructions in program?
  - # of cycles per second?
  - average # of cycles per instruction?
  - average # of instructions per second?

CPU Time Example

- **Example 1:**
  - CPU clock rate is 1 MHz
  - Program takes 45 million cycles to execute
  - What's the CPU time?

45,000,000 * (1 / 1,000,000) = 45 seconds

- **Example 2:**
  - CPU clock rate is 500 MHz
  - Program takes 45 million cycles to execute
  - What's the CPU time

45,000,000 * (1 / 500,000,000) = 0.09 seconds

CPI Example

- **Example:** Let assume that a benchmark has 100 instructions:
  - 25 instructions are loads/stores (each take 2 cycles)
  - 50 instructions are adds (each takes 1 cycle)
  - 25 instructions are square root (each takes 50 cycles)

What is the CPI for this benchmark?

$$CPI = ((0.25 * 2) + (0.50 * 1) + (0.25 * 50)) = 13.5$$

CPI Example

- Suppose we have two implementations of the same instruction set architecture (ISA).

  For some program,

  Machine A has a clock cycle time of 250 ps and a CPI of 2.0

  Machine B has a clock cycle time of 500 ps and a CPI of 1.2

  What machine is faster for this program, and by how much?

[10^{-3} = milli, 10^{-6} = micro, 10^{-9} = nano, 10^{-12} = pico, 10^{-15} = femto]

CPI Example

- Suppose we have two implementations of the same instruction set architecture (ISA).

  For some program,

  Machine A has a clock cycle time of **10 ns**. and a CPI of **2.0**
  Machine B has a clock cycle time of **20 ns**. and a CPI of **1.2**

- Which machine is faster for this program, and by how much?

Assume that # of instructions in the program is 1,000,000,000.

$\text{CPU Time}_A = 10^9 * 2.0 * 10 * 10^{-9} = 20$ seconds

$\text{CPU Time}_B = 10^9 * 1.2 * 20 * 10^{-9} = 24$ seconds

Machine A is faster

$$\frac{24}{20} = 1.2 \text{ times}$$

#Instructions Example

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles (respectively).

  The first code sequence has 5 instructions:
  2 of A, 1 of B, and 2 of C

  The second sequence has 6 instructions:
  4 of A, 1 of B, and 1 of C.

  Which sequence will be faster? How much?
  What is the CPI for each sequence?

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles (respectively).

The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C

The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.

- Which sequence will be faster? How much?
- What is the CPI for each sequence?

# of cycles for first code = (2 * 1) + (1 * 2) + (2 * 3) = 10 cycles

# of cycles for second code = (4 * 1) + (1 * 2) + (1 * 3) = 9 cycles

10 / 9 = 1.11 times

CPI for first code = 10 / 5 = 2

CPI for second code = 9 / 6 = 1.5

MIPS as performance measure

- A performance measure often used in practice to evaluate the performance of a computer system is the **MIPS rate** for a program A:

$$MIPS(A) = \frac{n_{instr}(A)}{T_{U\_CPU}(A) \cdot 10^6} \quad . \tag{1}$$

  $n_{instr}(A)$: number of instructions of program A
  $T_{U\_CPU}(A)$: user CPU time of program A

- modification:

$$MIPS(A) = \frac{r_{cycle}}{CPI(A) \cdot 10^6} \ ,$$

  where $r_{cycle} = 1/t_{cycle}$ is the clock rate of the processor.
  $CPI(A)$: **C**lock cycles **P**er **I**nstruction: average number of CPU cycles used for instructions of program A

- Faster processors lead to larger MIPS rates than slower processors.

MFLOPS as performance measure

- For program with scientific computations, the MFLOPS rate (**M**illion **F**loating-point **O**perations **P**er **S**econd) is sometimes used. The MFLOPS rate of a program A is defined by

$$MFLOPS(A) = \frac{n_{flp\_op}(A)}{T_{U\_CPU}(A) \cdot 10^6} \; [1/s] \; , \tag{2}$$

  $n_{flp\_op}(A)$: number of floating-point operations executed by A.
  $T_{U\_CPU}(A)$: user CPU time of program A

- The effective number of operations performed is used for MFLOPS: the MFLOPS rate provides a fair comparison of different program versions performing the same operations.

- Marketing metrics for computer performance included MIPS and MFLOPS
- MIPS : millions of instructions per second
  - MIPS = instruction count / (execution time x 10^6)
  - For example, a program that executes 3 million instructions in 2 seconds has a MIPS rating of 1.5
  - Advantage : Easy to understand and measure
  - Disadvantages : May not reflect actual performance, since simple instructions do better.
- MFLOPS : millions of floating point operations per second
  - MFLOPS = floating point operations / (execution time x 10^6)
  - For example, a program that executes 4 million fp. instructions in 5 seconds has a MFLOPS rating of 0.8
  - Advantage : Easy to understand and measure
  - Disadvantages : Same as MIPS, only measures floating point

- Two different compilers are being tested for a 500 MHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require one, two, and three cycles (respectively). Both compilers are used to produce code for a large piece of software.

  The first compiler's code uses 5 billions Class A instructions, 1 billion Class B instructions, and 1 billion Class C instructions.

  The second compiler's code uses 10 billions Class A instructions, 1 billion Class B instructions, and 1 billion Class C instructions.

- Which sequence will be faster according to MIPS?
- Which sequence will be faster according to execution time?

	Instruction counts (in billions) for each instruction class		
Code from	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

$$\text{CPU Clock cycles}_1 = (5 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 10 \times 10^9$$

$$\text{CPU Clock cycles}_2 = (10 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 15 \times 10^9$$

$$\text{CPU time}_1 = 10 \times 10^9 / 500 \times 10^6 = 20 \text{ seconds}$$

$$\text{CPU time}_2 = 15 \times 10^9 / 500 \times 10^6 = 30 \text{ seconds}$$

$$\text{MIPS}_1 = (5 + 1 + 1) \times 10^9 / 20 \times 10^6 = 350$$

$$\text{MIPS}_2 = (10 + 1 + 1) \times 10^9 / 30 \times 10^6 = 400$$

MIPS Example

- Two different compilers are being tested for a 4 GHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require one, two, and three cycles (respectively). Both compilers are used to produce code for a large piece of software.

  The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

  The second compiler's code uses 10 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

- Which sequence will be faster according to MIPS?
- Which sequence will be faster according to execution time?

Performance Summary

- The two main measure of performance are
  - execution time : time to do the task
  - throughput : number of tasks completed per unit time
- Performance and execution time are reciprocals. Increasing performance, decreases execution time.
- The time to execute a given program can be computed as:

  CPU time = Instruction count x CPI x clock cycle time

  CPU time = Instruction count x CPI / clock rate
- These factors are affected by compiler technology, the instruction set architecture, the machine organization, and the underlying technology.
- When trying to improve performance, look at what occurs frequently => make the common case fast.

- Performance evaluation is very important to assess programming quality as well as the underlying architecture and how they interact.
- The following capture some aspects of the system but do not represent overall performance: MIPS, #instructions, #cycles, frequency
- Execution time is what matters: system time, CPU time, I/O and memory time
- Scalability and efficiency measure the quality of your code.

Benchmark Programs

Different benchmark programs have been proposed for the evaluation of computer systems:

- **Synthetic benchmarks**
- **Kernel benchmarks**: small but relevant parts of real applications
- **Real application benchmarks** comprise several entire programs which reflect a workload of a standard user.
- popular benchmark suite: SPEC benchmarks (System Performance Evaluation Cooperation), see `www.spec.org`
- SPEC06 is the current version for desktop computers:
  12 integer programs (9 written in C, 3 in C++) and 17 floating-point programs (6 written in Fortran, 3 in C, 4 in C++, and 4 in mixed C and Fortran).

Benchmarks

- Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications
  - e.g., compilers/editors, scientific applications, graphics, etc.
- Small benchmarks
  - nice for architects and designers
  - easy to standardize
- Parallel Benchmarks: PARSEC, Rodinia, SPLASH-2
- SPEC (System Performance Evaluation Cooperative)
  - companies have agreed on a set of real program and inputs
  - valuable indicator of performance (and compiler technology)

Role of Benchmarks

- help designer explore architectural designs
- identify bottlenecks
- compare different systems
- conduct performance prediction

Example: PARSEC

- Princeton Application Repository for Shared-Memory Computers
- Benchmark Suite for Chip-Multiprocessors
- Freely available at: http://parsec.cs.princeton.edu/
- Objectives:
  - Multithreaded Applications: Future programs must run on multiprocessors
  - Emerging Workloads: Increasing CPU performance enables new applications
  - Diverse: Multiprocessors are being used for more and more tasks
  - State-of-Art Techniques: Algorithms and programming techniques evolve rapidly

Example: PARSEC

Program	Application Domain	Parallelization
Blackscholes	Financial Analysis	Data-parallel
Bodytrack	Computer Vision	Data-parallel
Canneal	Engineering	Unstructured
Dedup	Enterprise Storage	Pipeline
Facesim	Animation	Data-parallel
Ferret	Similarity Search	Pipeline
Fluidanimate	Animation	Data-parallel
Freqmine	Data Mining	Data-parallel
Streamcluster	Data Mining	Data-parallel
Swaptions	Financial Analysis	Data-parallel
Vips	Media Processing	Data-parallel
X264	Media Processing	Pipeline

Example: Rodinia

- A Benchmark Suite for Heterogeneous Computing: multicore CPU and GPU
- University of Virginia

Application / Kernel	Dwarf	Domain
K-means	Dense Linear Algebra	Data Mining
Needleman-Wunsch	Dynamic Programming	Bioinformatics
HotSpot*	Structured Grid	Physics Simulation
Back Propagation*	Unstructured Grid	Pattern Recognition
SRAD	Structured Grid	Image Processing
Leukocyte Tracking	Structured Grid	Medical Imaging
Breadth-First Search*	Graph Traversal	Graph Algorithms
Stream Cluster*	Dense Linear Algebra	Data Mining
Similarity Scores*	MapReduce	Web Mining

Estimating Perf. Improvements

- Assume a processor currently requires 10 seconds to execute a program and processor performance improves by 50 percent per year.
- By what factor does processor performance improve in 5 years?

  (1 + 0.5)^5 = 7.59
- How long will it take a processor to execute the program after 5 years?

  $ExTime_{new}$ = 10/7.59 = 1.32 seconds

Performance Example

- Computers M1 and M2 are two implementations of the same instruction set.
- M1 has a clock rate of 50 MHz and M2 has a clock rate of 100 MHz.
- M1 has a CPI of 2.8 and M2 has a CPI of 3.2 for a given program.
- How many times faster is M2 than M1 for this program?

$$\frac{\text{ExTime}_{M1}}{\text{ExTime}_{M2}} = \frac{\text{IC}_{M1} \times \text{CPI}_{M1} / \text{Clock Rate}_{M1}}{\text{IC}_{M2} \times \text{CPI}_{M2} / \text{Clock Rate}_{M2}} = \frac{2.8/50}{3.2/100} = 1.75$$

- What would the clock rate of M1 have to be for them to have the same execution time?

$$2.8 / \text{Clock Rate}_{M1} = 3.2 / 100 \implies \text{Clock Rate}_{M1} = 87.5 \text{ MHz}$$

For Multithreaded Programs

- Shall we use execution time or throughput? or both?
- IPC is not accurate here
  - small timing variations may lead to different execution
  - Order at which threads enter critical section may vary
  - Different interrupt timing may lead to different scheduling decisions

The total number of instructions executed may be different across different runs!

For Multithreaded Programs

The total number of instructions executed may be different across different runs!

This effect increases with the number of cores

System-level code account for a significant fraction of the total execution time

Your Program Does Not Run in A Vacuum

- System software at least is there
- Multi-programming and/or mulithreading setting is very common in multicore settings
- Independent programs affect each other performance (why?)

Standard Definition of Performance

- For some program running on machine X,

$$\text{Performance}_X = 1 / \text{Execution time}_X$$

- "X is n times faster than Y"

$$\text{Performance}_X \ / \ \text{Performance}_Y = n$$

  - Example: time taken to run a program
    - 10s on A, 15s on B
    - Execution Time_B / Execution Time_A = 15s / 10s = 1.5
    - So A is 1.5 times faster than B

Speedup

- Number of cores = p
- Serial run-time = T_{serial}
- Parallel run-time = $T_{parallel}$

$$S = \frac{T_{serial}}{T_{parallel}}$$

Efficiency of a parallel program

$$E = \frac{S}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}}\right)}{p}$$

Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*.

- **Strong scaling:**
  - The total problem size stays fixed as more processors are added.
  - Goal is to run the same problem size faster
  - Perfect scaling means problem is solved in 1/P time (compared to serial)

- **Weak scaling:**
  - The problem size *per processor* stays fixed as more processors are added. The total problem size is proportional to the number of processors used.
  - Goal is to run larger problem in same amount of time
  - Perfect scaling means problem Px runs in same time as single processor run

- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more processors is rarely the answer.

- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. This is a common situation with many parallel applications.

- Hardware factors play a significant role in scalability. Examples:
  - Memory-cpu bus bandwidth on an SMP machine
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed

- Parallel support libraries and subsystems software can limit scalability independent of your application.

- **Amdahl's Law** states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

$$\text{speedup} = \frac{1}{1 - P}$$

- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
- If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.
- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

where P = parallel fraction, N = number of processors and S = serial fraction.

- Assume that a program runs in 100 seconds on a machine, with multiply operations responsible for 80 seconds. How much do I have to improve the speed of multiplication if I want my program to run 2 times faster.

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

$$50 \text{ seconds} = \frac{80 \text{ seconds}}{n} + (100 - 80 \text{ seconds})$$

$$n = \frac{80 \text{ seconds}}{30 \text{ seconds}} = 2.67$$

- Speedup due to an enhancement is defined as:

$$\text{Speedup} = \frac{\text{ExTime old}}{\text{ExTime new}} = \frac{\text{Performance new}}{\text{Performance old}}$$

- Suppose that an enhancement accelerates a fraction
- $\textbf{Fraction}_{\textbf{enhanced}}$ of the task by a factor $\textbf{Speedup}_{\textbf{enhanced}}$

$$\textbf{ExTime}_{\textbf{new}} = \textbf{ExTime}_{\textbf{old}} \times \left[(1 - \textbf{Fraction}_{\textbf{enhanced}}) + \frac{\textbf{Fraction}_{\textbf{enhanced}}}{\textbf{Speedup}_{\textbf{enhanced}}} \right]$$

$$\textbf{Speedup} = \frac{\textbf{ExTime}_{\textbf{old}}}{\textbf{ExTime}_{\textbf{new}}} = \frac{1}{(1 - \textbf{Fraction}_{\textbf{enhanced}}) + \frac{\textbf{Fraction}_{\textbf{enhanced}}}{\textbf{Speedup}_{\textbf{enhanced}}}}$$

Proof :-

Let Speedup be S, old execution time be T, new execution time be T' , execution time that is taken by portion A(that will be enhanced) is t, execution time that is taken by portion A(after enhancing) is t', execution time that is taken by portion that won't be enhanced is t_n, Fraction enhanced is f', Speedup enhanced is S'.

Now from the above equation,

$$S = \frac{T}{T'}$$

$$T = t_n + t$$

$$T' = t_n + t'$$

$$f' = \frac{t}{T}$$

$$= \frac{t}{t+t_n}$$

$$1 - f' = 1 - \frac{t}{t+t_n}$$

$$= \frac{t_n}{t+t_n}$$

$$S' = \frac{t}{t'}$$

$$t' = \frac{t}{s'}$$

$$= \frac{T*f'}{S'}$$

$$= \frac{(t_n+t)*f'}{S'}$$

$$= \frac{t'}{t_n+t} = \frac{f'}{S'}$$

$$S = \frac{T}{T'}$$

$$= \frac{t_n+t}{t_n+t'}$$

$$S = \frac{1}{1-f'+\frac{f'}{S'}}$$

$$Overall Speedup = \frac{1}{1 - Fraction Enhanced + (Fraction Enhanced / Speedup Enhanced)}$$

Hence proved.

- **Amdahl's Law**

Suppose that we are considering an enhancement that runs 10 times faster than the original machine but is usable only 40% of the time. What is the overall speedup gained by incorporating the enhancement?

S=1/((1-F)+F/K)

=1/(0.6+0.4/10) = 1/0.64 = 1.56

- Floating point instructions are improved to run twice as fast, but only 10% of the time was spent on these instructions originally. How much faster is the new machine?

$$\text{Speedup} = \frac{\text{ExTime}_{old}}{\text{ExTime}_{new}} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

$$\text{Speedup} = \frac{1}{(1 - 0.1) + 0.1/2} = 1.053$$

  - The new machine is 1.053 times as fast, or 5.3% faster.
  - How much faster would the new machine be if floating point instructions become 100 times faster?

$$\text{Speedup} = \frac{1}{(1 - 0.1) + 0.1/100} = 1.109$$

Amdhal's effect

- Ramifications to Amdhal's law: Many problems have the property that the inherently sequential part of the solution is a linear function of problem size, n.
- Notice that the maximum speedup for a larger problem size is larger
- Amdahl's Law does not account for the Parallel overhead, a large part of which are the communication costs
- As the problem size increased, the fraction of the computation that was inherently sequential decreased. This is often the case. Furthermore, the parallel overhead often has smaller complexity as a function of n. than the parallelizable portion of the computation. A consequence of these facts is that, as the problem size increases, for a fixed number of. processors, the maximum possible speedup tends to increase. This relationship has been called the Amdahl effect.
- The speedup is critically dependent on the degree to which the program or the application is inherently sequential

- Amdahl’s Law states a limit on just how much faster a parallel program can run than a given sequential program for a problem of a fixed size

- When the number of processors is increased, the problem size is increased linearly with it and the running time of the parallel version of the program remains fixed, not the problem size.

- Gustafson's Law:
  - Speedup with P processors is

$S(P)=P-\alpha\,(P-1) = \alpha+P(1-\alpha)$

  - Amdahl's law assume that the entire problem is of fixed size so that the total amount of work to be done in parallel is also independent of the number of the processors.
  - Gustafson's law assumes that the total amount of work to be done in parallel varies linearly with the number of processors.

Types of communications

- **Persistent Communication**: a message that has been submitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver.
- In this case, the middleware will store the message at one or several of the storage facilities.
- No need of time coupling between sender and receiver
- **Transient Communication**: A Message is stored by the communication system only as long as the sending and receiving application are executing.
- Communication system consists of traditional store and forward routers, if cannot, it will drop the message
- Time coupling between sender and the receiver is required

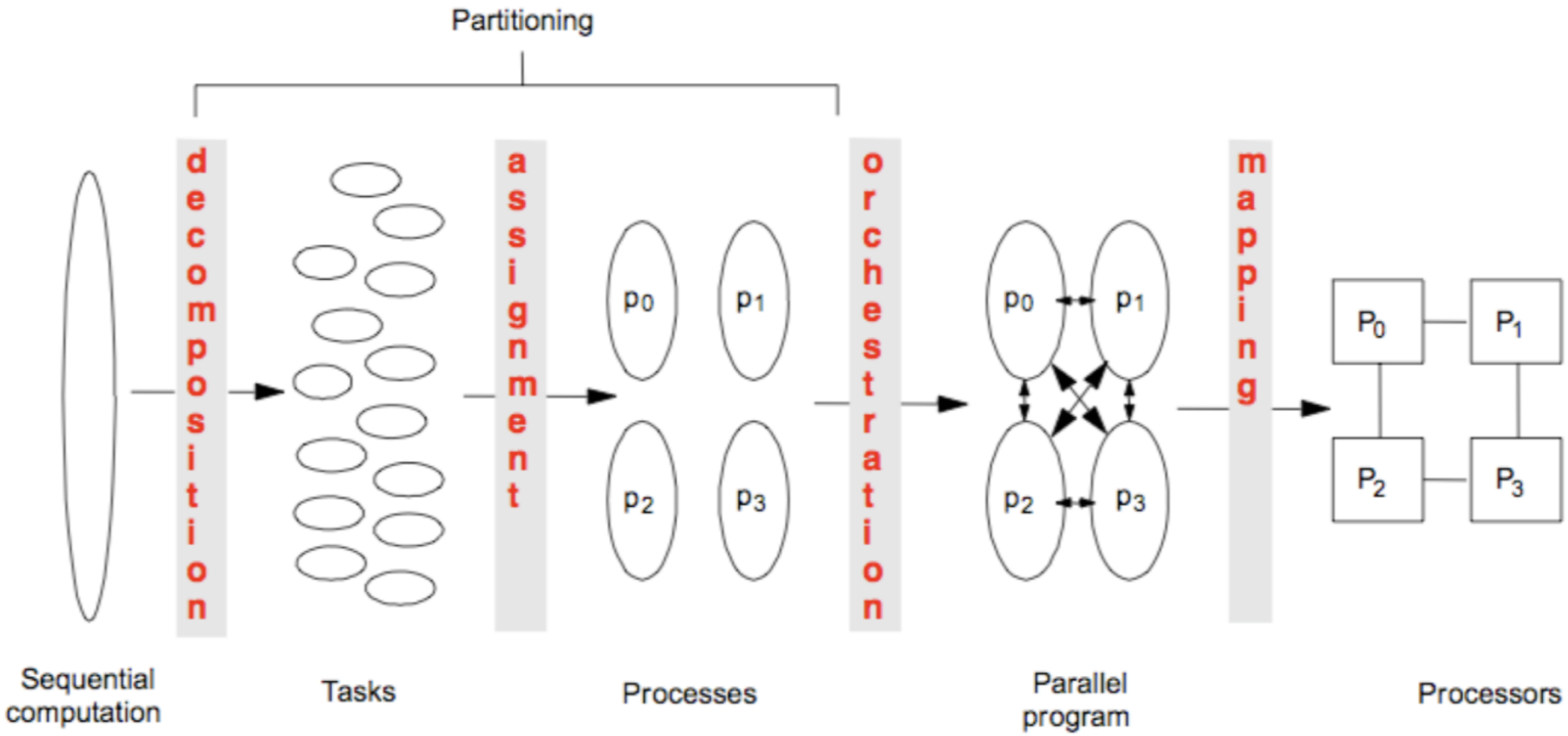
Types of Communications

- **Asynchronous communication**: Sender continues immediately after it has submitted its message for transmission (message gets stored)
- **Synchronous communication**: The sender is blocked until its request is known to be accepted
- Three stages of synchronization: sender may block until middleware notifies that it will deliver the message or receiver has received a message or until the receiver sends a response.
- Many combinations are possible
- Persistent and synchronous is more common in message-queueing systems

Types of communications

- **Discrete Communication**: Most common form, the parties communicate by messages, each message forming a complete unit of information
- **Streaming communication**: It involves sending multiple messages, one after the other where the messages are related to each other by the order they are sent or because there is a temporal relationship. Example: RPC

Parallel Algorithm Design

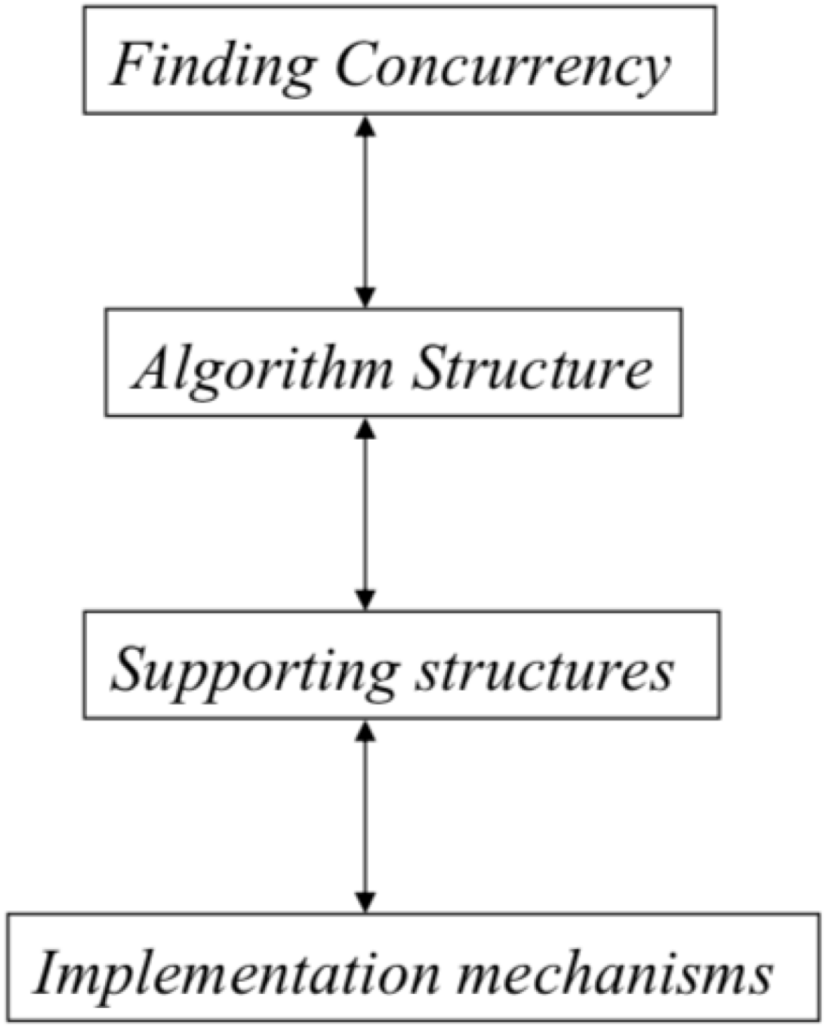


1. Study problem or code

2. Look for parallelism opportunities

3. Try to keep all cores busy doing useful work

Parallel Algorithm Design



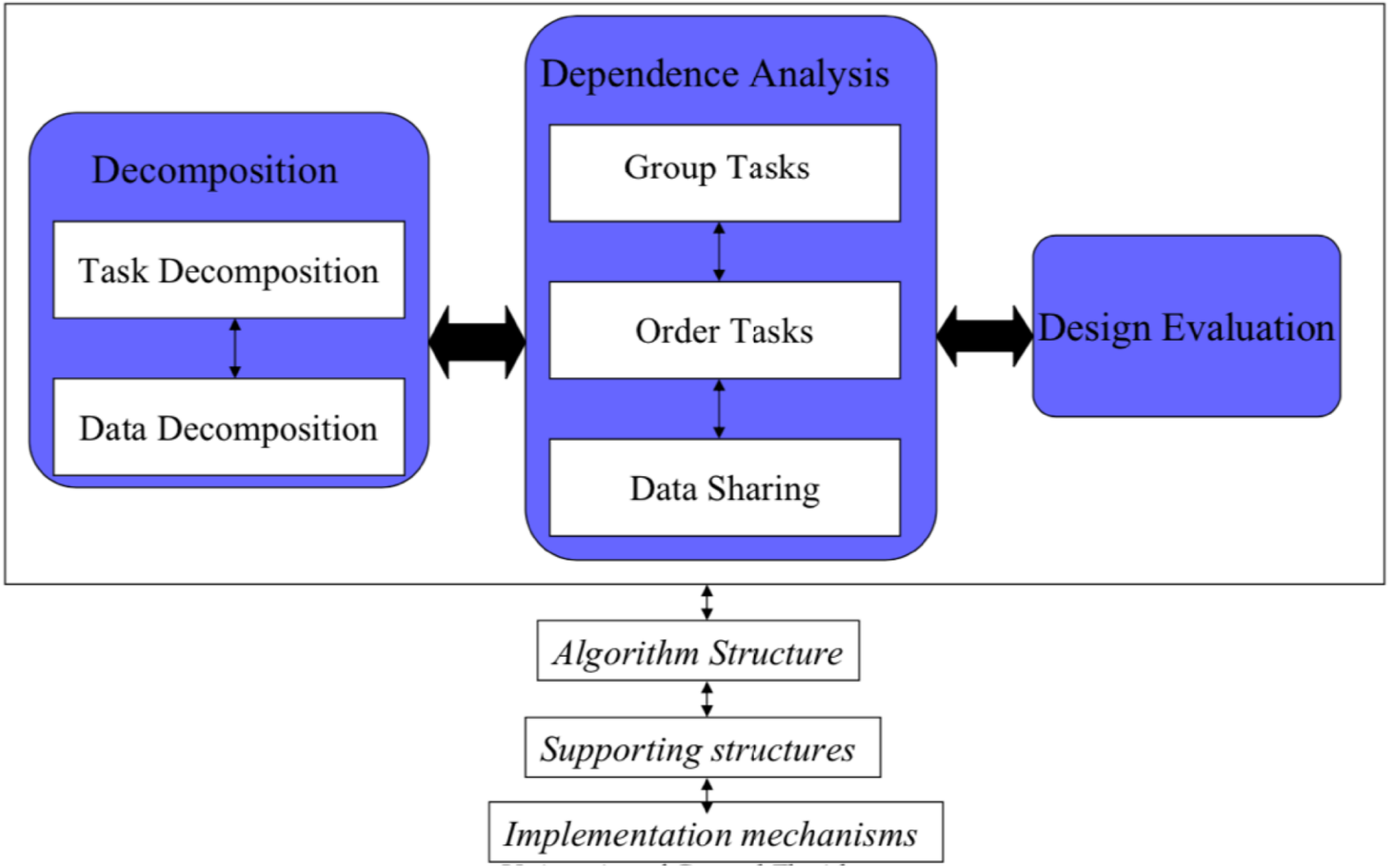
Exposing exploitable concurrency

Structuring algorithm to take advantage

Structuring program and shared data

Mapping to particular programming environment

Finding Concurrency



Decomposition Patterns

- Task decomposition: view problem as a stream of instructions that can be broken into sequences called tasks that can execute in parallel.
  - Key: Independent operations

- Data decomposition: view problem from data perspective and focus on how the can be broken into distinct chunks
  - Key: Data chunks that can be operated upon independently

- Task and data decomposition imply each other. They are different facets of the same fundamental decomposition

Example

- Matrix multiplication
  - Task decomposition
    - Considering the computation of each element in the product matrix as a separate task
    - Performs poorly => group tasks pattern
  - Data decomposition
    - Decompose the product matrix into chunks, e.g., one row a chunk, or a small submatrix (or block) per chunk

Dependency Analysis Pattern

- Group tasks: group tasks that have the same dependency constraints; identify which tasks must execute concurrently
  - Reduced synchronization overhead – all tasks in the group can use a barrier to wait for a common dependence
  - All tasks in the group efficiently share data loaded into a common on-chip, shared storage (Shard Memory)
  - Grouping and merging dependent tasks into one task reduces need for synchronization

- Order task pattern: identifying order constraints among task groups.
  - Control dependency: Find the task group that creates it
  - Data dependency: temporal order for producer and consumer relationship

Dependency Analysis Pattern

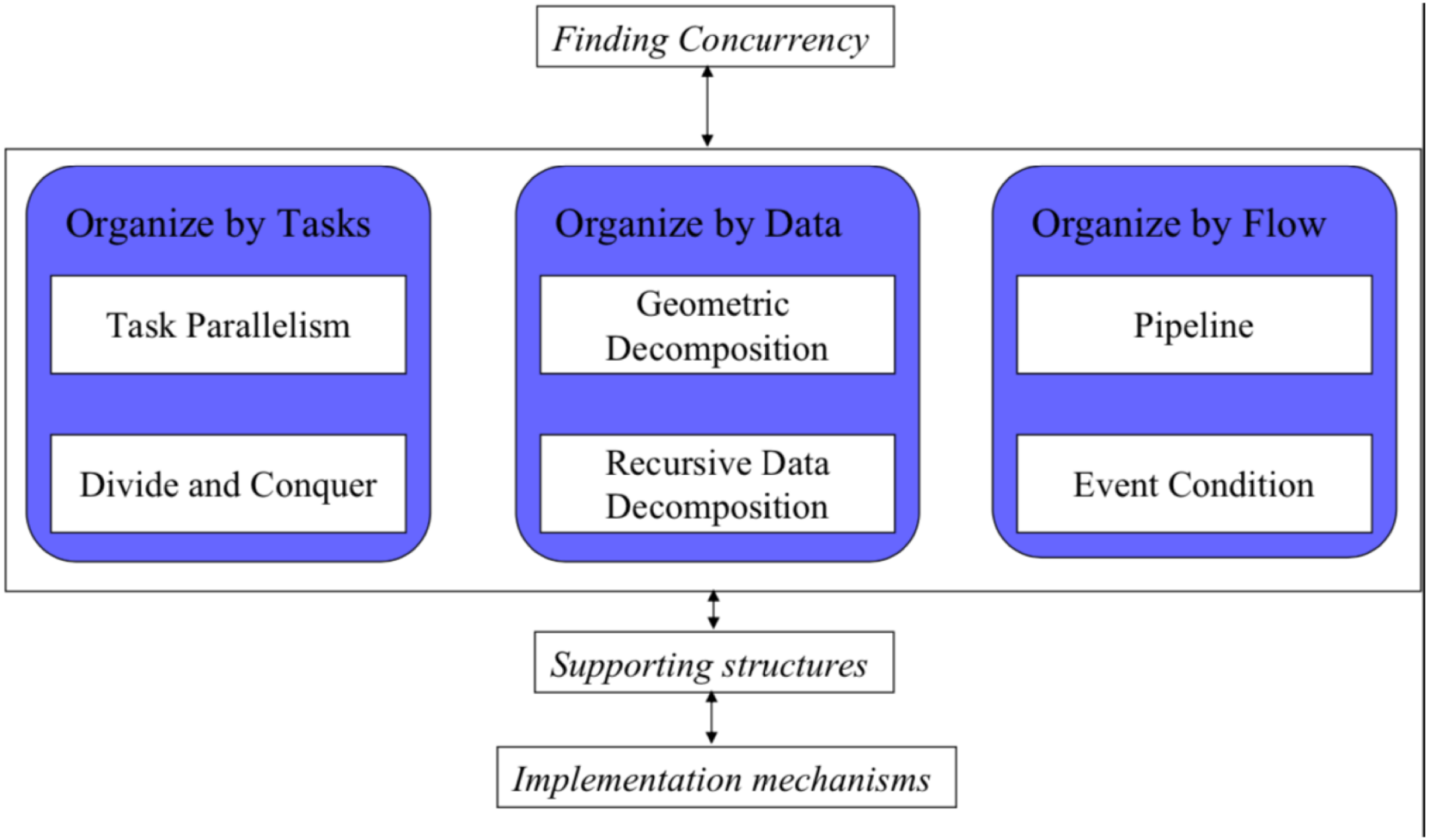
- Data sharing pattern: how data is shared among the tasks?
  - Read only: make own local copies
  - Effectively local: the shared data is partitioned into subsets, each of which is accessed (for read or write) by only one task a time.
  - Read-write: the data is accessed by more than one task. Need exclusive access mechanisms.
  - Example: the use of the shared memory among threads in a thread block.

Design Evaluation Pattern

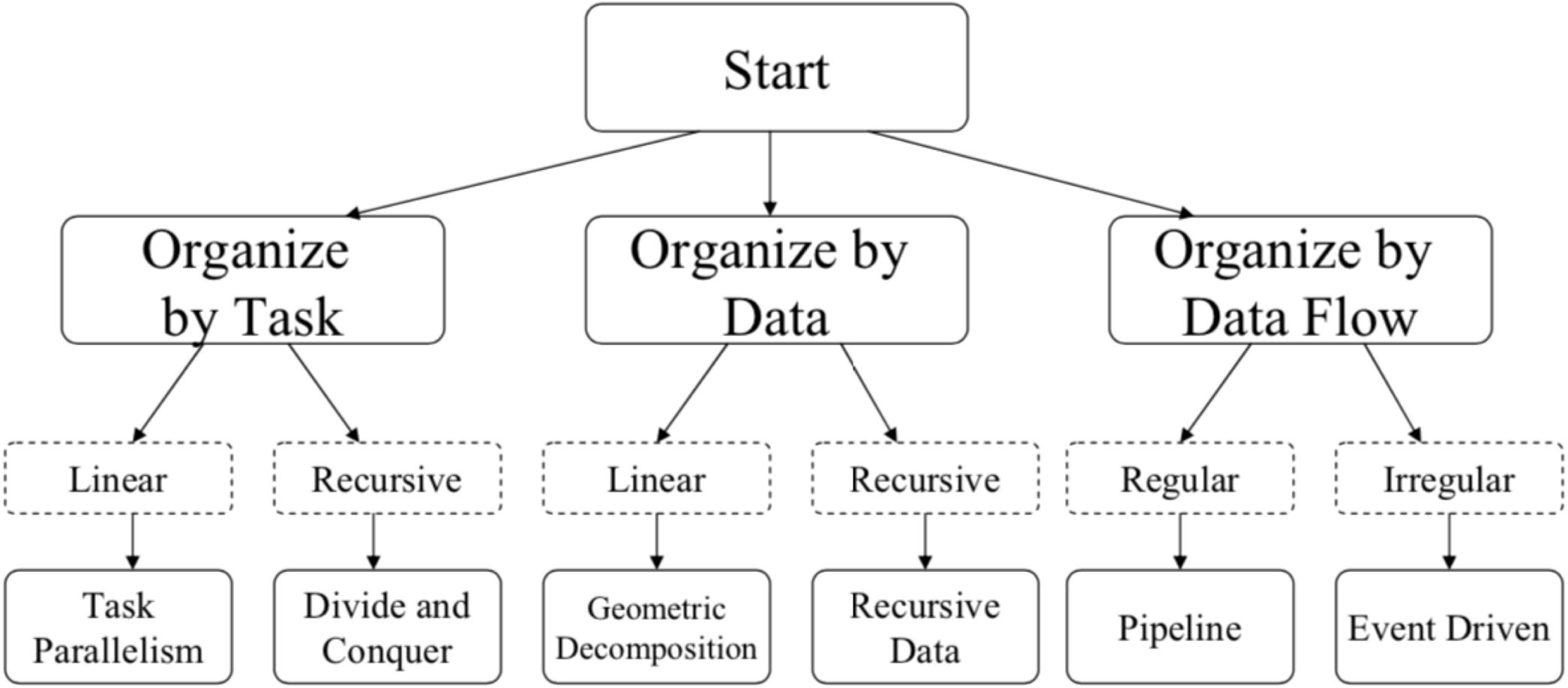
- Whether the partition fits the target hardware platform?

- Key questions to ask
  - How many threads can be supported?
  - How many threads are needed?
  - How are the data structures shared?
  - Is there enough work in each thread between synchronizations to make parallel execution worthwhile?

Algorithm Structure



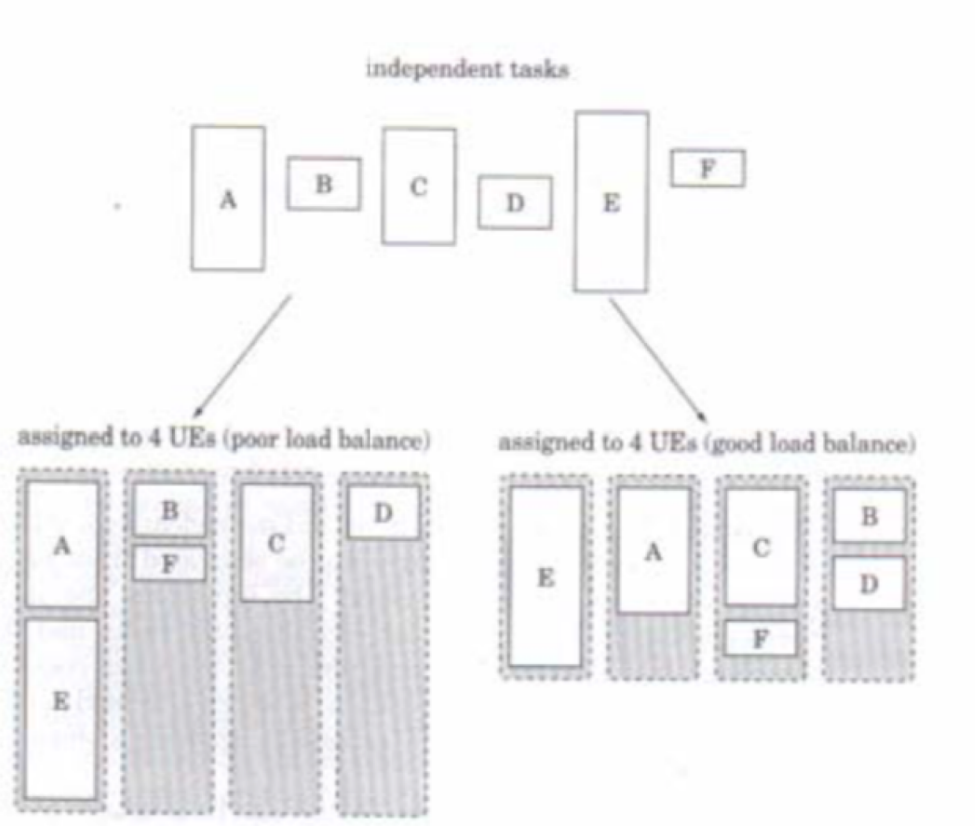
Organizing Principle



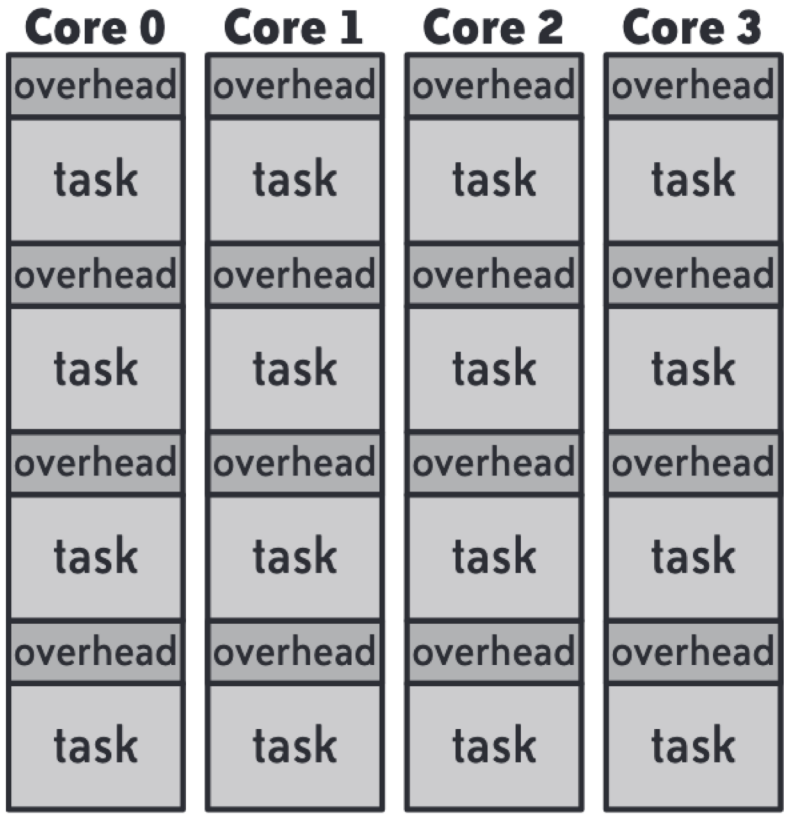
Task Parallelism Pattern

- After the problem is decomposed into a collection of tasks that can execute concurrently, how to exploit this concurrency efficiently?

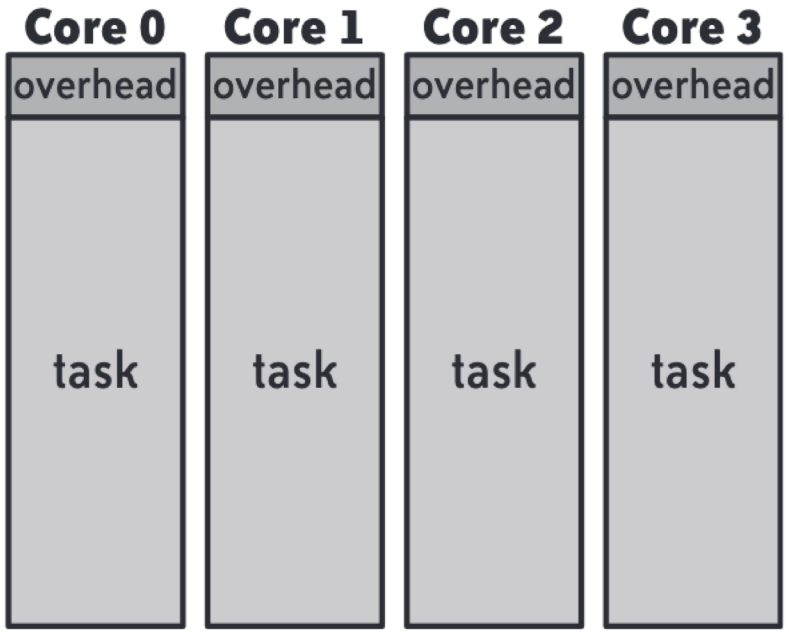
- Load balancing



Task Granularity Example



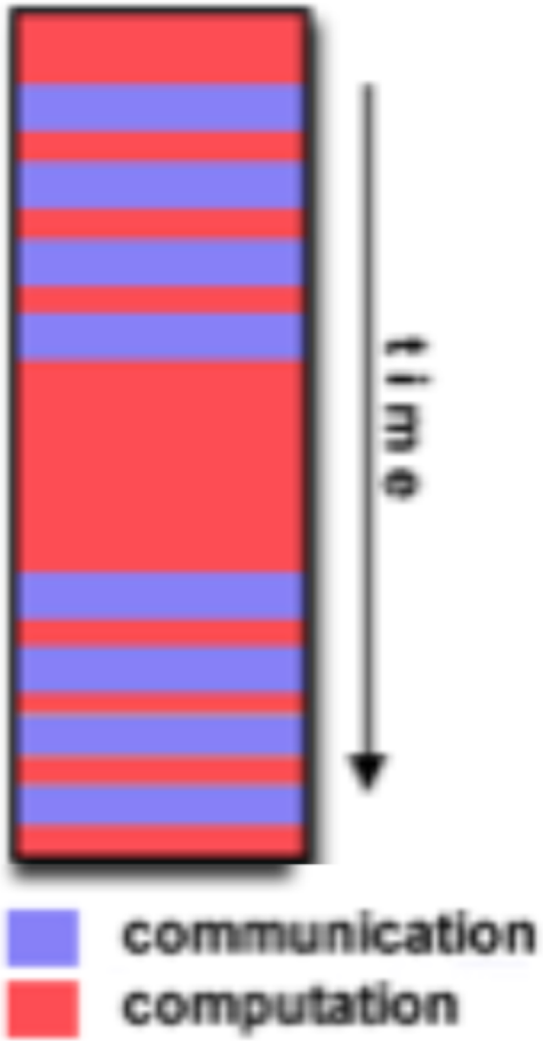
(a) Fine-grained decomposition



(b) Coarse-grained decomposition

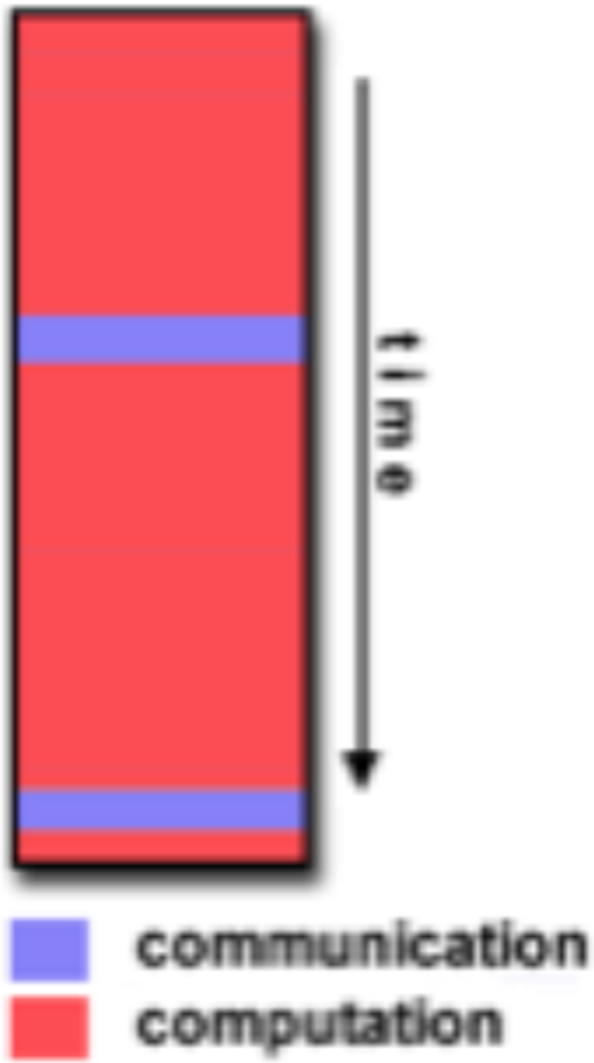
Task Granularity Example

- Tasks execute little comp. between comm.
- Easy to load balance
- If ***too fine***, comm. may take longer than comp.



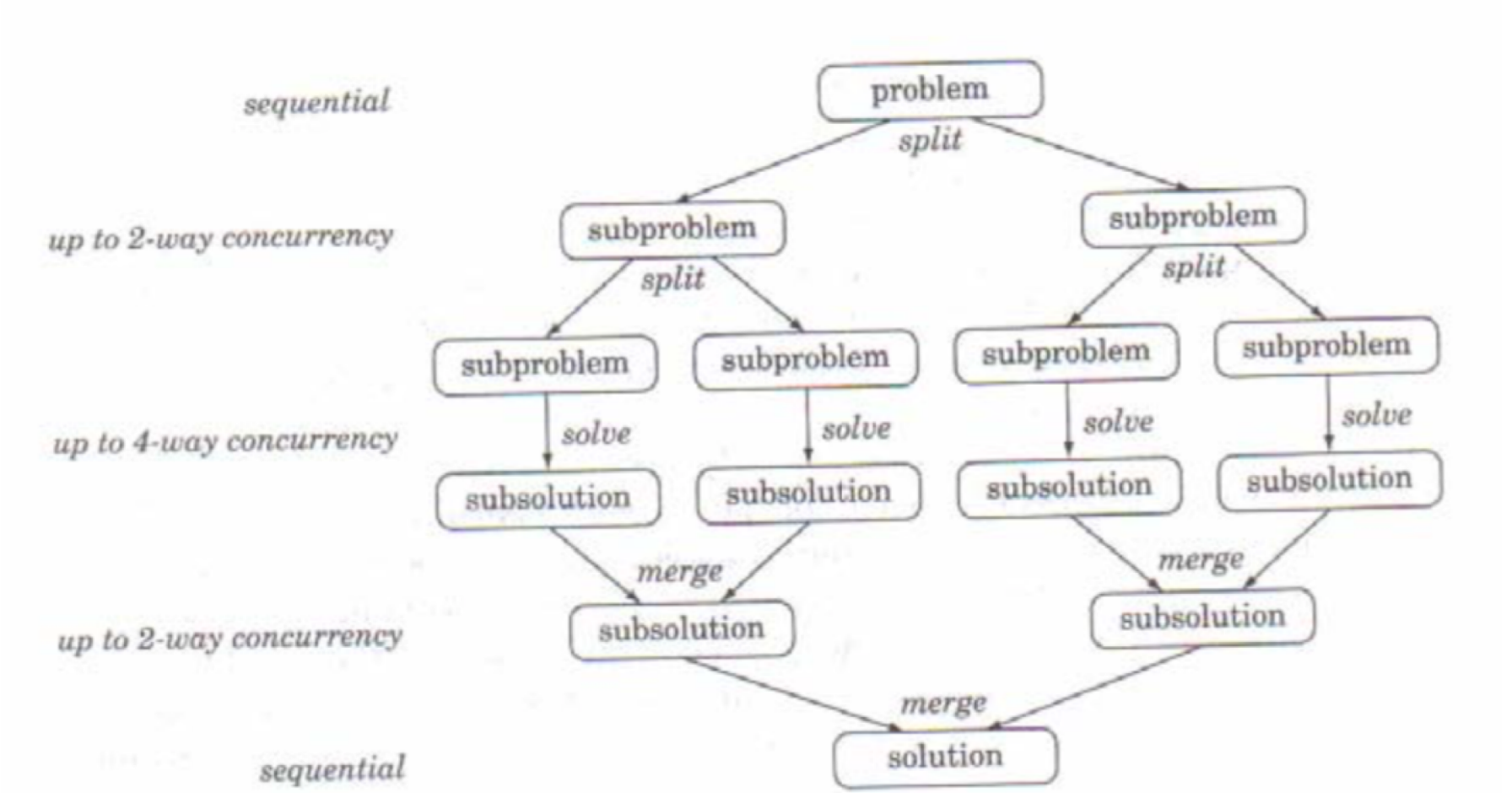
Task Granularity Example

- Long computations between communication
- More opportunity for performance increase
- Harder to load balance



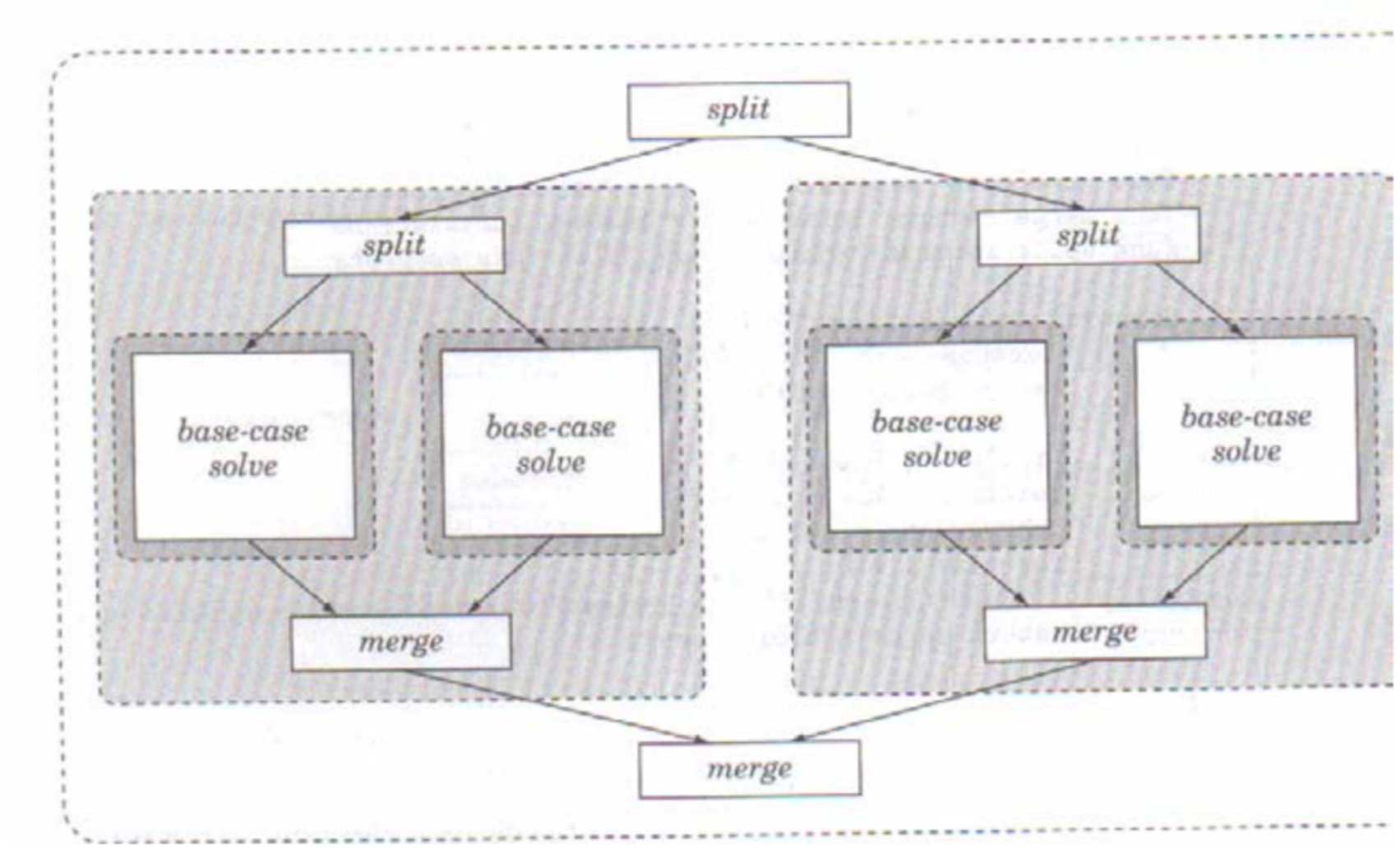
Divide and Conquer Pattern

- If the problem is formulated using the sequential divide-and-conquer strategy, how to exploit the potential concurrency?



Divide and Conquer Pattern

- Parallelization Strategy



Geometric Decomposition Pattern

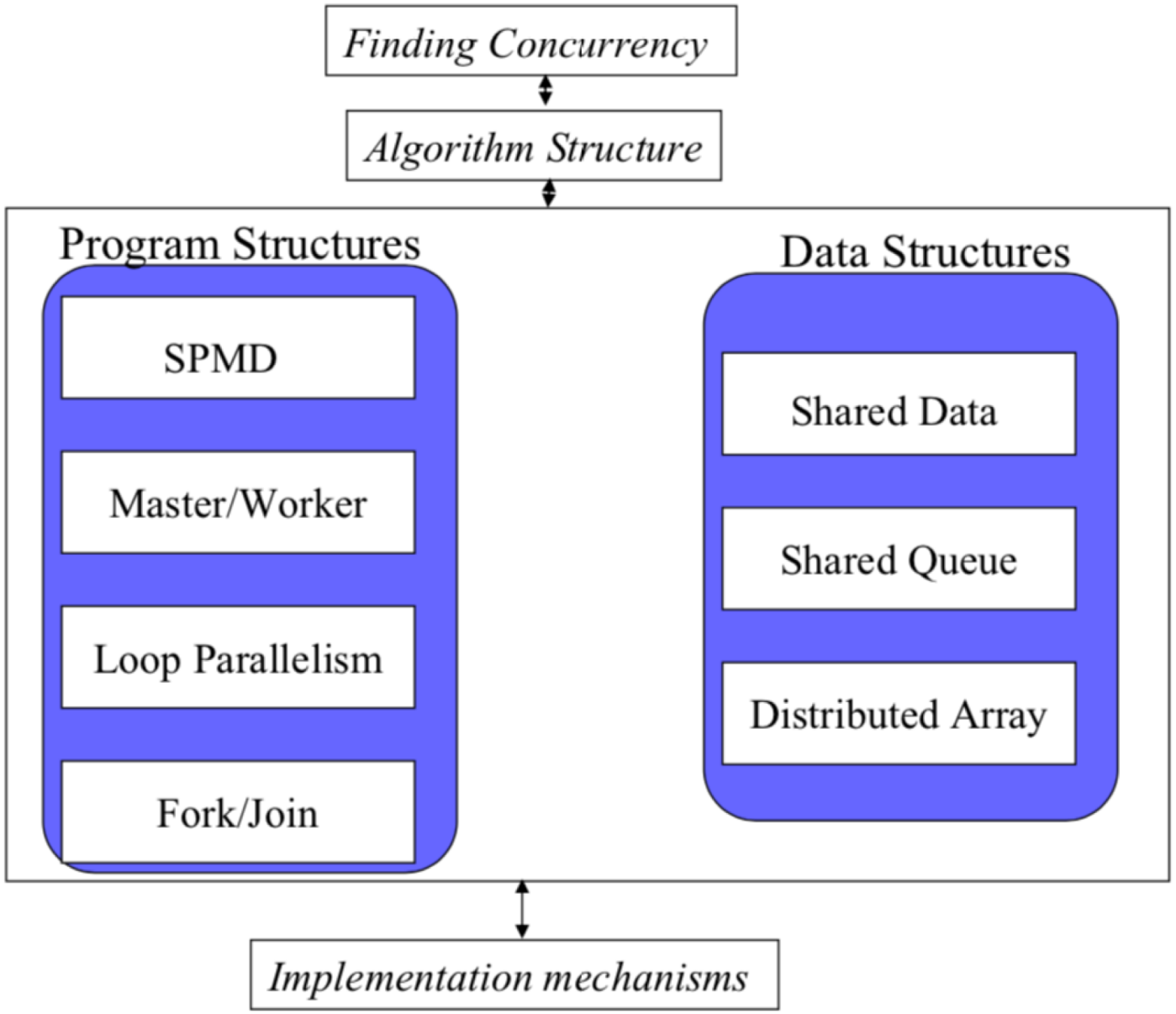
- How to organize the algorithm after the data has been decomposed into concurrently updatable chunks?

- Decomposition to minimize the data communication and dependency among tasks
- Care needs to be taken when update non-local data, e.g., exchange operations

Recursive Data Pattern

- Suppose the problem involves an operation on a recursive data structure that appears to require sequential processing. How to make the operations on these data structures parallel?
- Check whether divide-and-conquer pattern works
- If not, may need to transform the original algorithm.

Supporting Structures



Relationship between supporting program structure pattern and Algorithm structure pattern

	Task Parallel.	Divide/Conquer	Geometric Decomp.	Recursive Data	Pipeline	Event-based
SPMD	☺ ☺ ☺ ☺	☺ ☺ ☺	☺ ☺ ☺ ☺	☺ ☺	☺ ☺ ☺	☺ ☺
Loop Parallel	☺ ☺ ☺ ☺	☺ ☺	☺ ☺ ☺			
Master/Worker	☺ ☺ ☺ ☺	☺ ☺	☺	☺	☺	☺
Fork/Join	☺ ☺	☺ ☺ ☺ ☺	☺ ☺		☺ ☺ ☺ ☺	☺ ☺ ☺ ☺

Relationship between supporting program structure pattern and programming environment

	Task Parallel.	Divide/Conquer	Geometric Decomp.	Recursive Data	Pipeline	Event-based
SPMD	☺ ☺ ☺ ☺	☺ ☺ ☺	☺ ☺ ☺ ☺	☺ ☺	☺ ☺ ☺	☺ ☺
Loop Parallel	☺ ☺ ☺ ☺	☺ ☺	☺ ☺ ☺			
Master/Worker	☺ ☺ ☺ ☺	☺ ☺	☺	☺	☺	☺
Fork/Join	☺ ☺	☺ ☺ ☺ ☺	☺ ☺		☺ ☺ ☺ ☺	☺ ☺ ☺ ☺

Implementation Mechanism