departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concurrency and Parallelism
(Concorrência e Paralelismo – CP 11158)

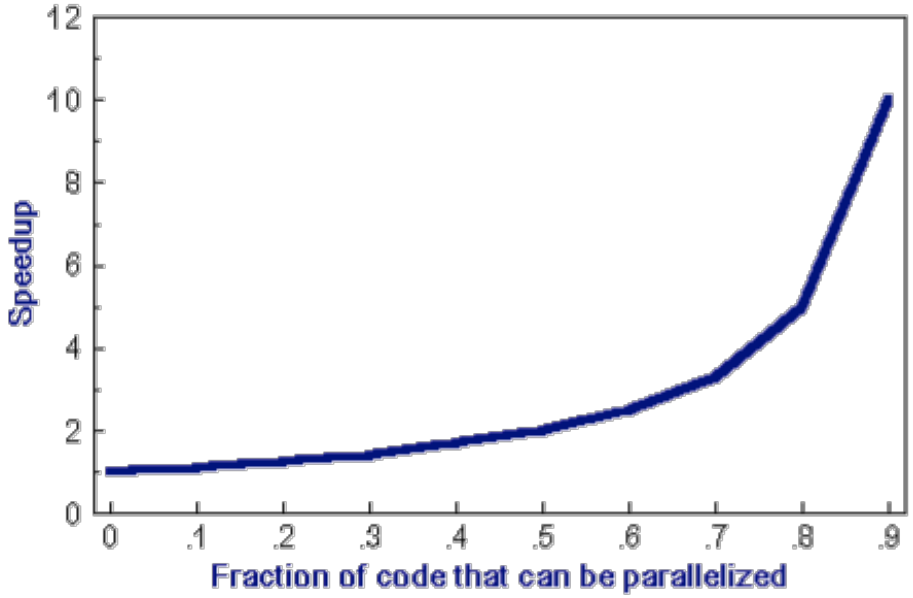
Lecture 11

— Map-Reduce —

Amdahl's Law

- 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.

Amdahl's Law

- Introducing the number of processors performing the parallel fraction of work, the relationship can be modelled by

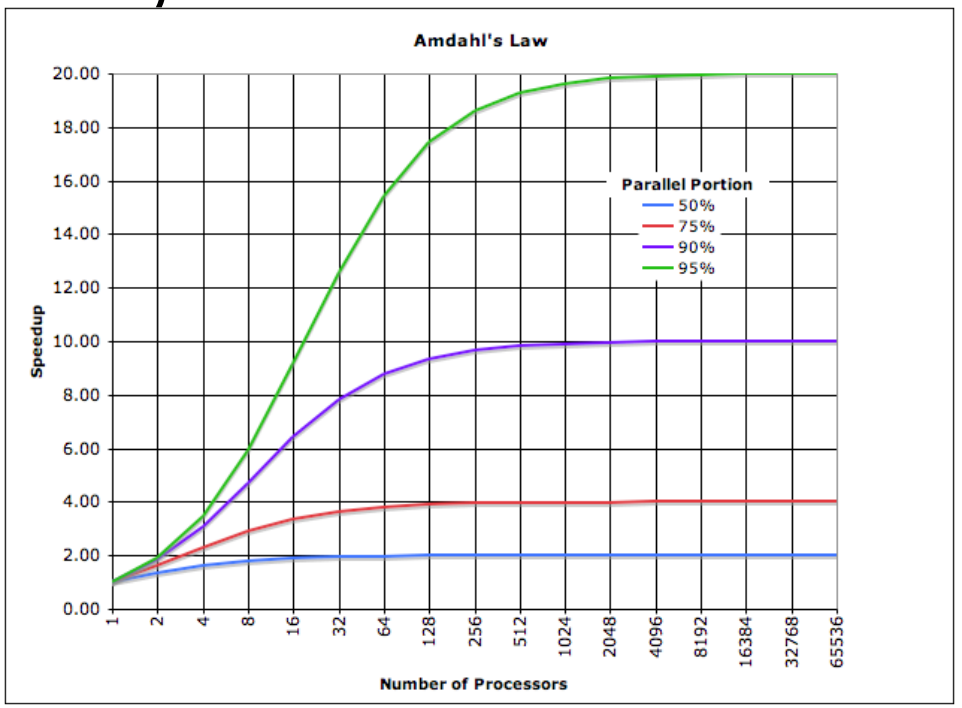
$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

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

Amdahl's Law

- It soon becomes obvious that there are limits to the scalability of parallelism. For example, at P = .50, .90 and .99 (50%, 90% and 99% of the code is parallelizable)

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

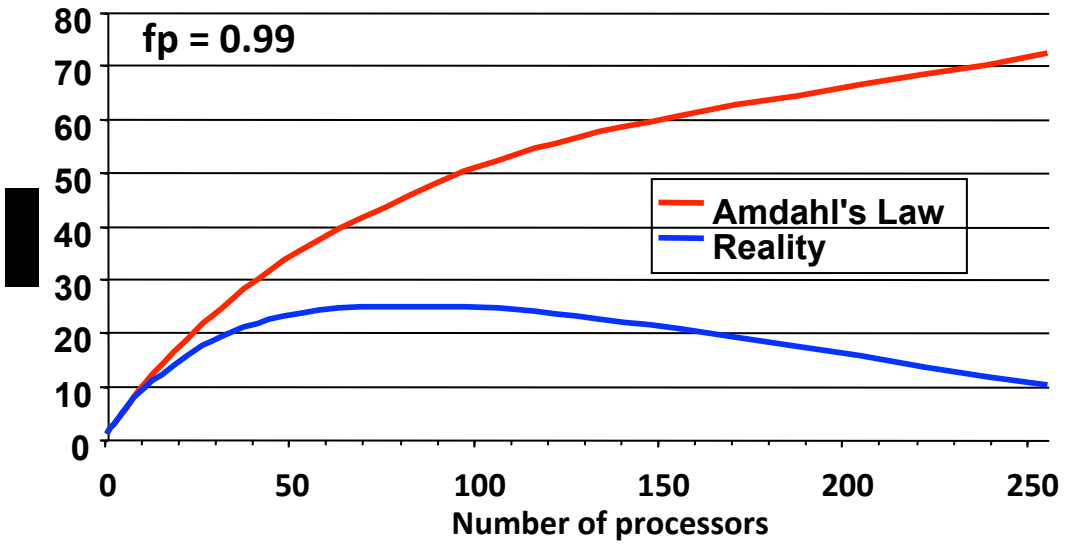


Amdahl's Law

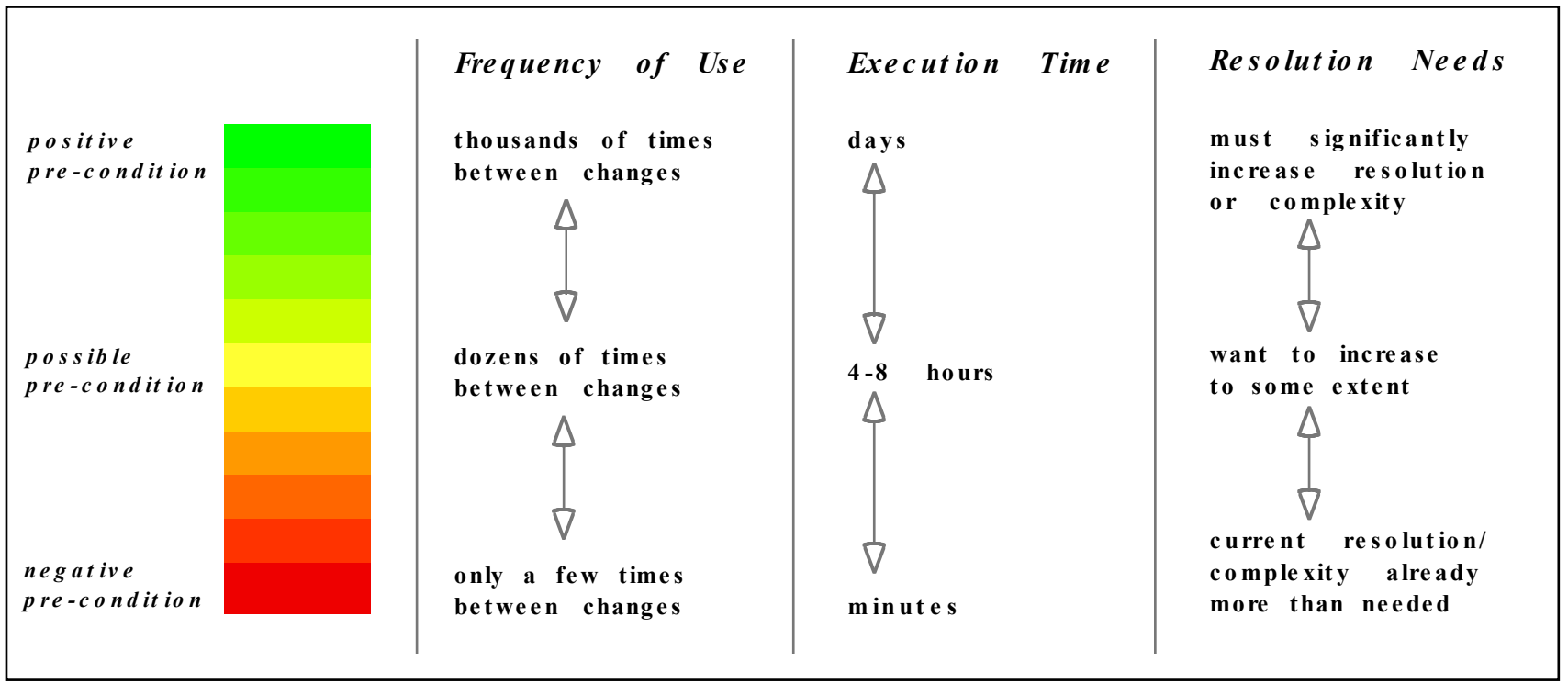
- However, certain problems demonstrate increased performance by increasing the problem size. For example:
  - **2D Grid Calculations 85 seconds 85%**
  - **Serial fraction 15 seconds 15%**
- We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:
  - **2D Grid Calculations 680 seconds 97.84%**
  - **Serial fraction 15 seconds 2.16%**
- Problems that increase the percentage of parallel time with their size are more ***scalable*** than problems with a fixed percentage of parallel time.

Amdahl's Law Vs. Reality

- Amdahl's Law provides a theoretical upper limit on parallel speedup assuming that there are no costs for communications.
  - In reality, communications will result in a further degradation of performance.



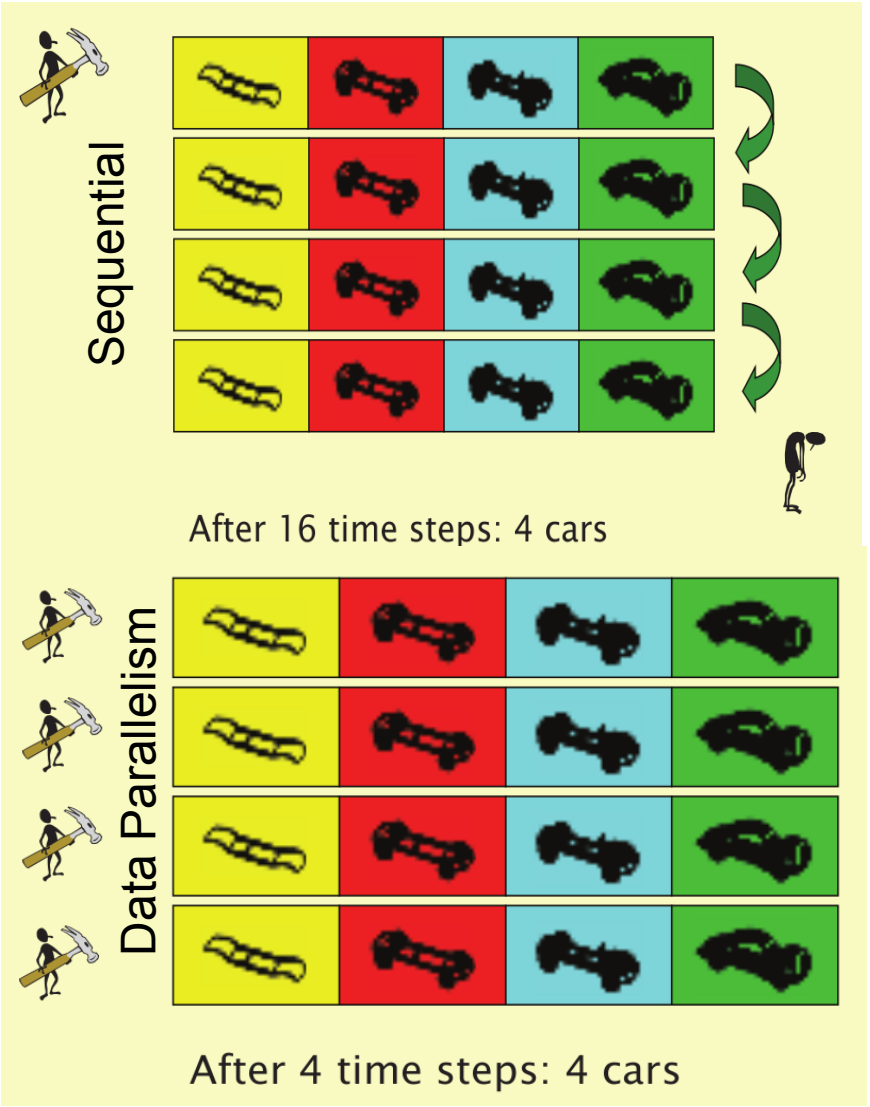
Test the "Preconditions for Parallelism"



- According to experienced parallel programmers:
  - *no green* – Don't even consider it
  - *one or more red* – Parallelism may cost you more than you gain
  - *all green* – You need the power of parallelism (but there are no guarantees)

Parallelism — — A simplistic understanding

- Multiple tasks at once.
- Distribute work into multiple execution units.
- A classification of parallelism:
  - Data Parallelism
  - Functional or Control Parallelism
- Data Parallelism
  - Divide the dataset and solve each sector "similarly" on a separate execution unit.
- Functional Parallelism
  - Divide the 'problem' into different tasks and execute the tasks on different units.
  - What would func. parallelism look like for the example on the right?



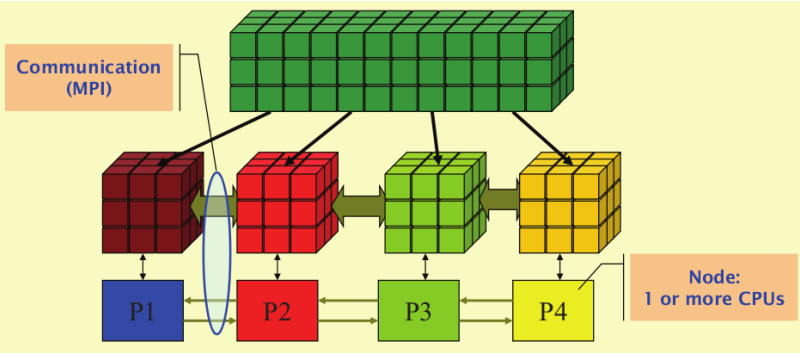
Partitioning

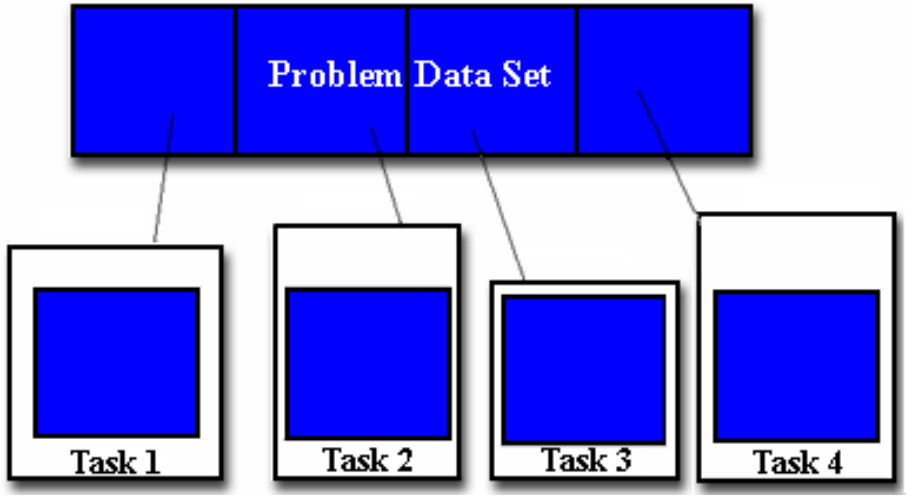
- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as **decomposition** or **partitioning**.
- There are two basic ways to partition computational work among parallel tasks:
  - ***Domain decomposition***

    and
  - ***Functional decomposition***

Domain Decomposition

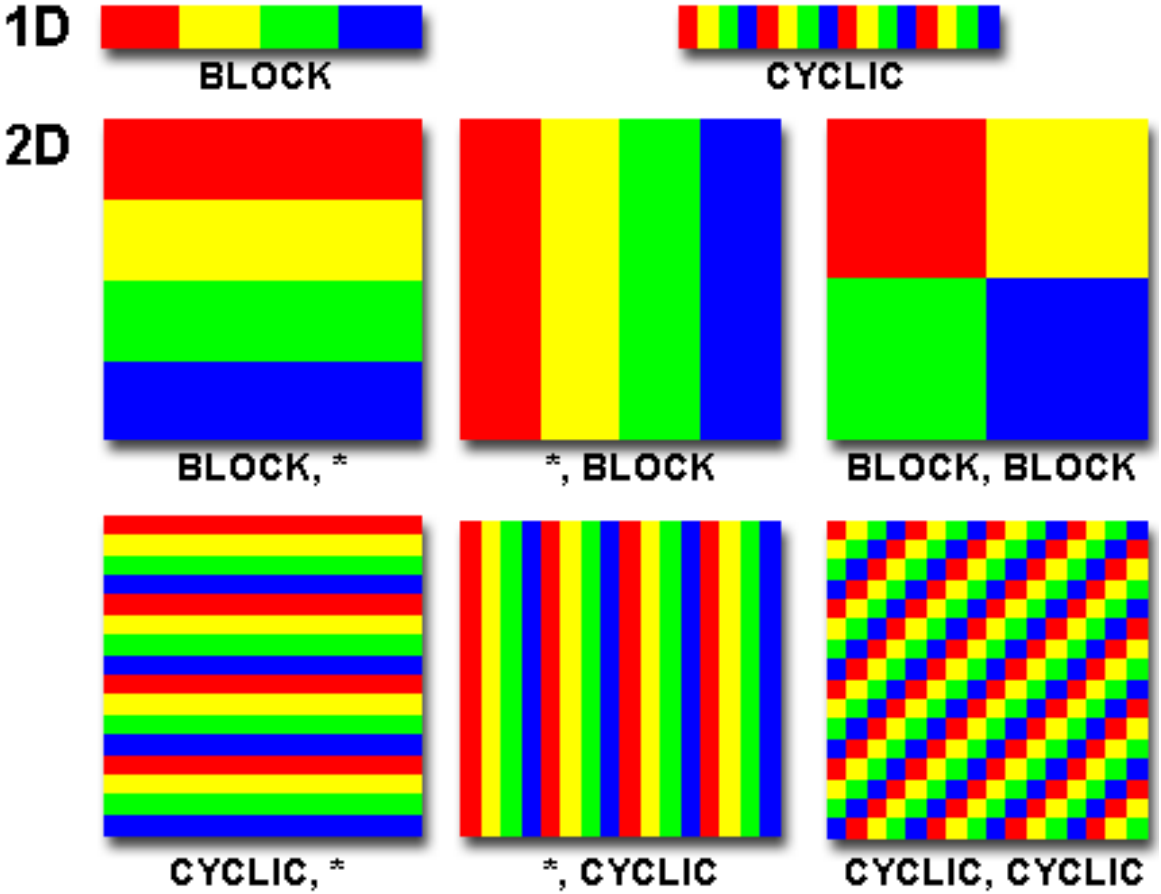
- In this type of partitioning, the data associated with a problem is decomposed.
- Each parallel task then works on a portion of of the data.





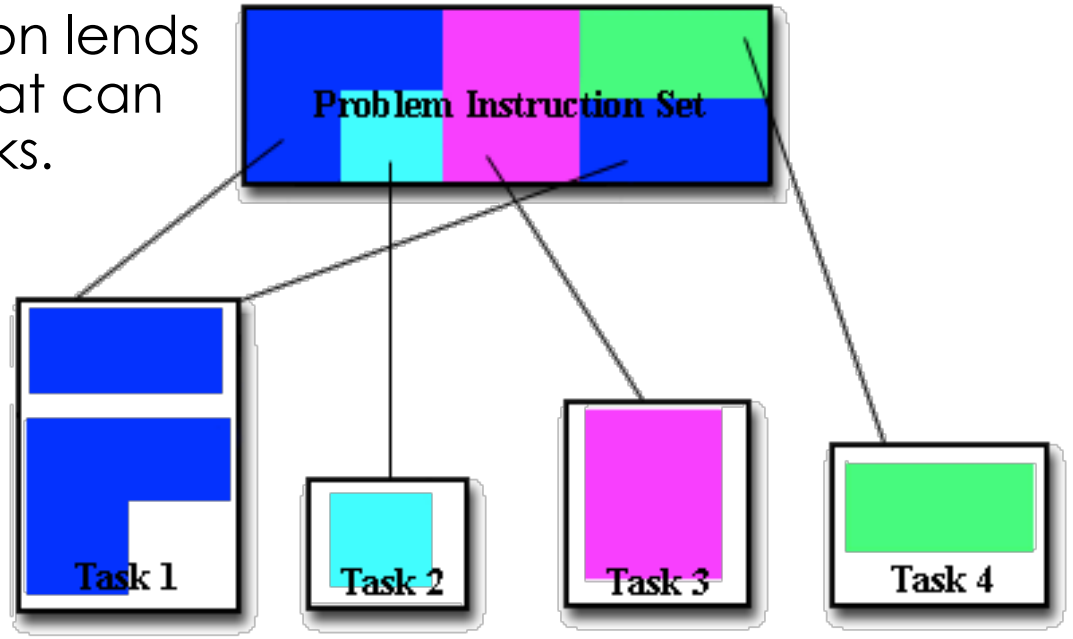
Partitioning Data

- There are different ways to partition data



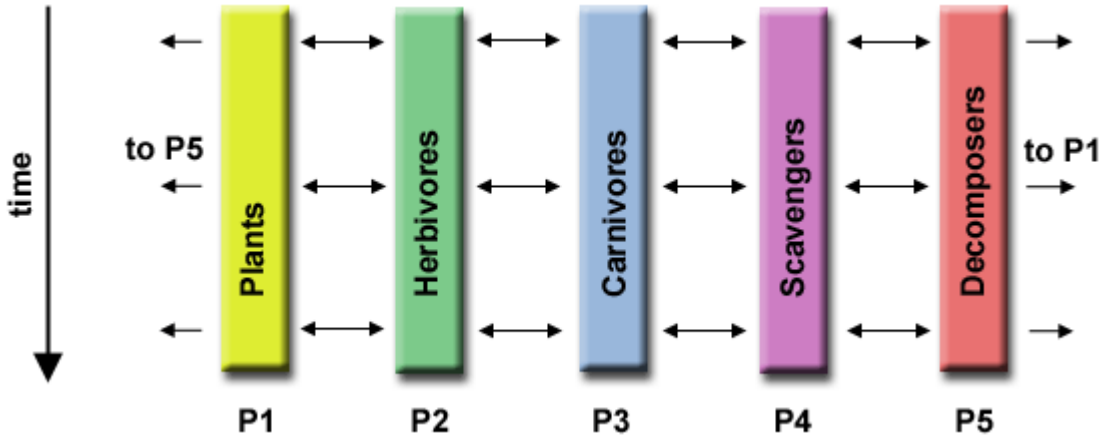
Functional Decomposition

- The focus is on the computation rather than on the data manipulated by the computation.
- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
- Functional decomposition lends itself well to problems that can be split into different tasks.
- For example
  - Ecosystem Modeling
  - Signal Processing
  - Climate Modeling



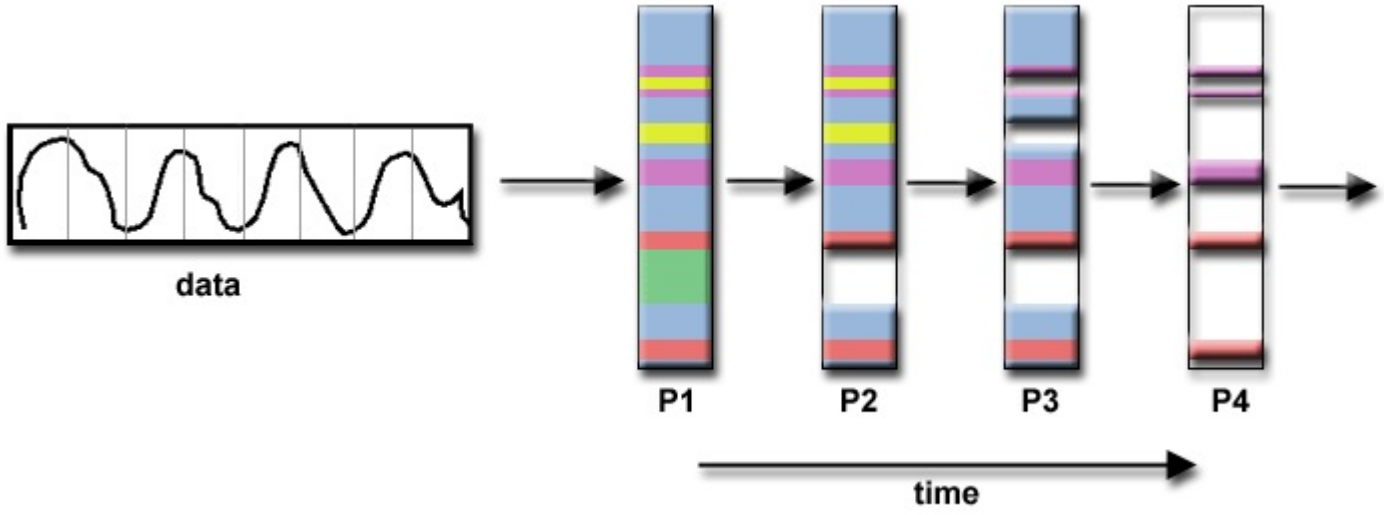
Ecosystem Modeling

- Each program calculates the population of a given group, where each group's growth depends on that of its neighbours.
  - As time progresses, each process calculates its current state
  - Then exchanges information with the neighbour populations
  - All tasks then progress to calculate the state at the next time step.



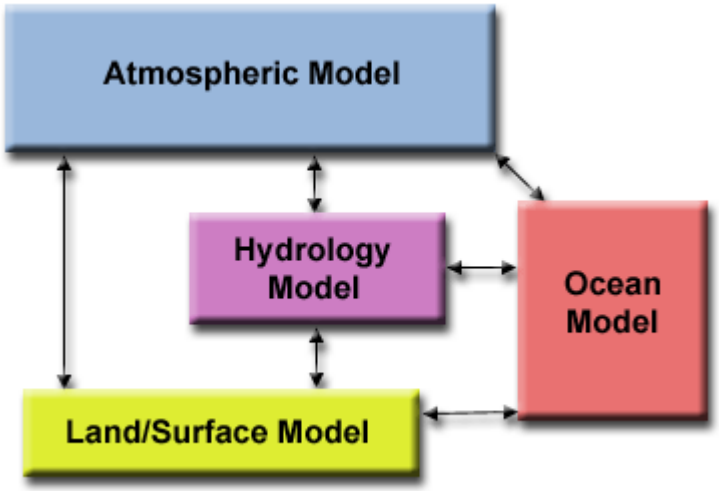
Signal Processing (pipelining)

- An audio signal data set is passed through four distinct computational filters.
  - Each filter is a separate process.
  - The first segment of data must pass through the first filter before progressing to the second.
  - When it does, the second segment of data passes through the first filter.
  - By the time the fourth segment of data is in the first filter, all four tasks are busy.



Climate Modeling

- Each model component can be thought of as a separate task.
- Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.



- Combining these two types of problem decomposition is common and natural.

Superlinear Speedup

- S(n) > n, may be seen on occasion
  - due to using a suboptimal sequential algorithm; or
  - some unique feature of the architecture that favors the parallel formation, e.g.
    - extra memory in the multiprocessor system which can hold more of the problem data at any instant.
    - Improved cache locality effect.

Efficiency

$$\text{Efficiency} = \frac{\text{Execution time using one processor}}{\text{Execution time using a number of processors}}$$

$$= \frac{t_s}{t_p \times n}$$

Its just the speedup divided by the number of processors

$$E = \frac{S(n)}{n} \times 100\%$$

Cost

The processor-time product or cost (or work) of a computation defined as
Cost = (execution time) x (total number of processors used)

The cost of a sequential computation is simply its execution time, t_s . The cost of a parallel computation is t_p x n. The parallel execution time, t_p , is given by $t_s/S(n)$

Hence, the cost of a parallel computation is given by

$$\text{Cost} = \frac{t_s n}{S(n)} = \frac{t_s}{E}$$

Cost-Optimal Parallel Algorithm
One in which the cost to solve a problem on a multiprocessor is proportional to the cost

Scalability

- Used to indicate a hardware design that allows the system to be increased in size and in doing so to obtain increased performance
  - Could be described as architecture or hardware scalability.
- Scalability is also used to indicate that a parallel algorithm can accommodate increased data items with a low and bounded increase in computational steps
  - could be described as algorithmic scalability.

Problem size

Problem size: the number of basic steps in the **best sequential algorithm** for a given problem and data set size

Intuitively, we would think of the number of data elements being processed in the algorithm as a measure of size.

However, doubling the date set size would not necessarily double the number of computational steps. It will depend upon the problem.

For example, adding two matrices has this effect, but multiplying matrices quadruples operations.

Note: Bad sequential algorithms tend to scale well

Gustafson's law

Rather than assume that the problem size is fixed, assume that the parallel execution time is fixed. In increasing the problem size, Gustafson also makes the case that the serial section of the code does not increase as the problem size.

Scaled Speedup Factor

The scaled speedup factor becomes

$$S_s(n) = \frac{s + np}{s + p} = s + np = n + (1 - n)s$$

called Gustafson's law.

Example
Suppose a serial section of 5% and 20 processors; the speedup according to the formula is 0.05 + 0.95(20) = 19.05 instead of 10.26 according to Amdahl's law. (Note, however, the different assumptions.)

Parallel Programming Issues

- The main goal of writing a parallel program is to get better performance over the serial version. Several issues that you need to consider:
  - Load balancing
  - Minimizing communication
  - Overlapping communication and computation

Load Balancing

- **Load balancing** is the task of equally dividing work among the available processes.
- This can be easy to do when the same operations are being performed by all the processes (on different pieces of data).
- When there are large variations in processing time, you may be required to adopt a different method for solving the problem.

Minimizing Communication

- Total execution time is a major concern in parallel programming because it is an essential component for comparing and improving all programs.
- Three components make up execution time:
  - Computation time
  - Idle time
  - Communication time

Minimizing Communication

- **Computation time** is the time spent performing computations on the data.
- **Idle time** is the time a process spends waiting for data from other processors.
- Finally, **communication time** is the time it takes for processes to send and receive messages.
  - The cost of communication in the execution time can be measured in terms of latency and bandwidth.
  - **Latency** is the time it takes to set up the envelope for communication, where **bandwidth** is the actual speed of transmission, or bits per unit time.
  - Serial programs do not use inter-process communication. Therefore, you must minimize this use of time to get the best performance improvements.

Overlapping Communication and Computation

- There are several ways to minimize idle time within processes, and one example is overlapping communication and computation. This involves occupying a process with one or more new tasks while it waits for communication to finish so it can proceed on another task.
- Careful use of nonblocking communication and data unspecific computation make this possible. It is very difficult in practice to interleave communication with computation.

departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

The End