Designing Parallel Programs

Dra. Mireya Paredes López
mireya.paredes@udlap.mx

Automatics vs. Manual Parallelization

- Designing and developing parallel programs has been a very ***manual process***.
- Manually parallelization → Time consuming, complex, error-prone and iterative process.
- Available tools to ***assist*** the programmer → compiler/pre-processor.

Automatics vs. Manual Parallelization

Fully Automatic

- The compiler analyzes the source code and *identifies opportunities for parallelism*.

- The analysis includes identifying ***inhibitors*** to parallelism.

- The compiler finds out if parallelism would actually improve performance.

- Loops are the *most frequent* target for automatic parallelization.

Automatics vs. Manual Parallelization

Programmer Directed

- Using "compiler directives" or possibly ***compiler flags***, the programmer explicitly tells the compiler how to parallelize the code.

- May be able to be used in conjunction with some degree of ***automatic parallelization*** also.

Automatic Parallelization problems

- Wrong results may be produced.
- Performance may actually degrade.
- Much less flexible than manual parallelization.
- Limited to a subset (mostly loops) of code.
- May actually not parallelize code if the compiler analysis suggest that there are ***inhibitors / too complex.***

Understand the Problems and the Program

- First → to understand the problem that you wish to solve in parallel.
- Before spending time in an attempt to develop a **parallel solution for a problem** → Determine if the problem can actually ***be parallelized***.

Example:

Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.

Understand the Problems and the Program

- Example of a problem with little-to-no parallelism.

  Example:

  Calculation of the Fibonacci series (0, 1, 1, 2, 3,5, 8, 13, 21, ……) by use of the formula:

  $F(n) = F(n-1) + F(n-2)$

  The calculation of the F(n) value uses those of both F(n-1) and F(n-2), which must be computed first.

Understand the Problems and the Program

Identify the program´s **hotspots:**

- Know where most of the ***real work*** is being done.
- ***Profilers*** and ***performance analysis tools*** can help here.
- Focus on ***parallelizing the hotspots*** and ignore those sections of the program that account for little CPU usage.
- s

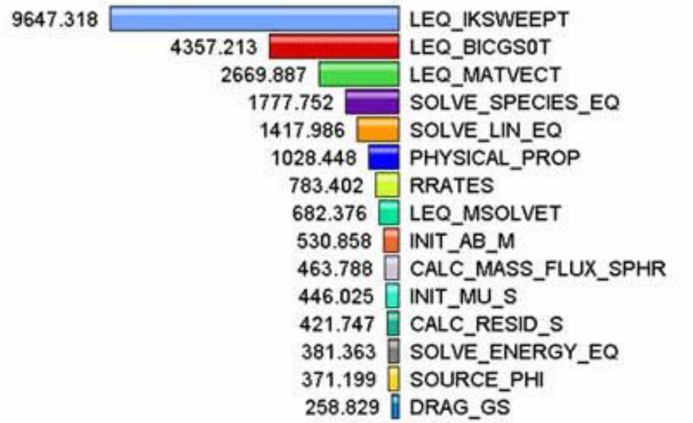
Understand the Problems and the Program

Identify **bottlenecks** in the program:

- Are there areas that are disproportionately slow?
- For example, I/O is usually something that slows a program down.
- May be possible to reestructure the program.

Understand the Problems and the Program

HOTSPOTS



BOTTLENECK

Understand the Problems and the Program

Identify inhibitors to parallelism. → ***data dependence***

Investigate other algorithms if possible.

Take advantage **of optimized third party parallel software** → highly ***optimized math libraries*** available from vendors (IBM, Intel, AMD, etc.).

Partitioning

- *To break the problem into* discrete "chunks" of work that can be distributed to multiple tasks. → **decomposition**

- There are two basic ways to partition computational work among parallel tasks →

partitioning

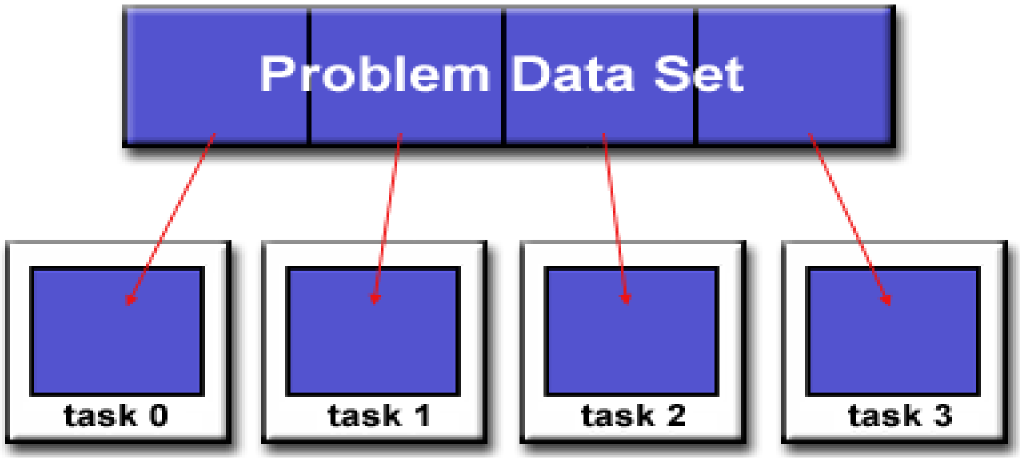
Domain decomposition

Functional decomposition

Domain Decomposition

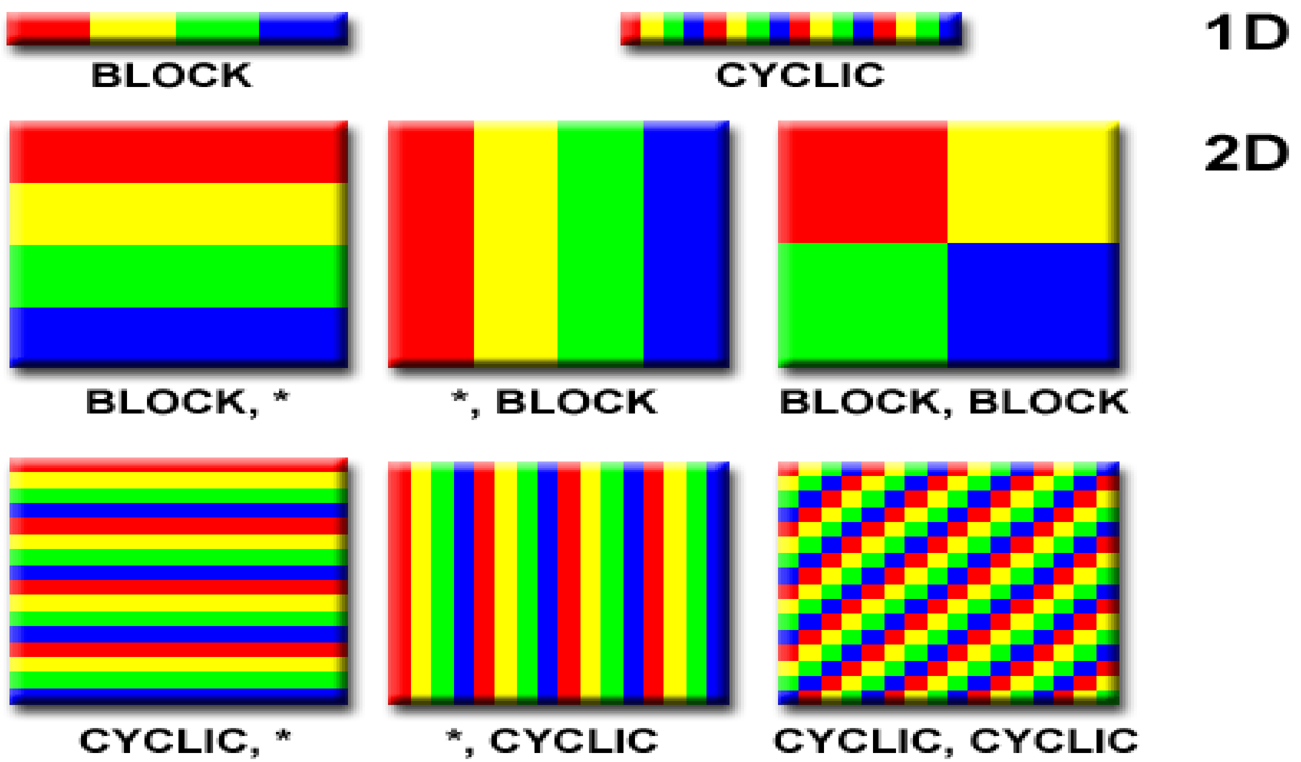
In this type of partitioning, the **data** associated with a problem is ***decomposed***.

- Each parallel task then works on a portion of the data.



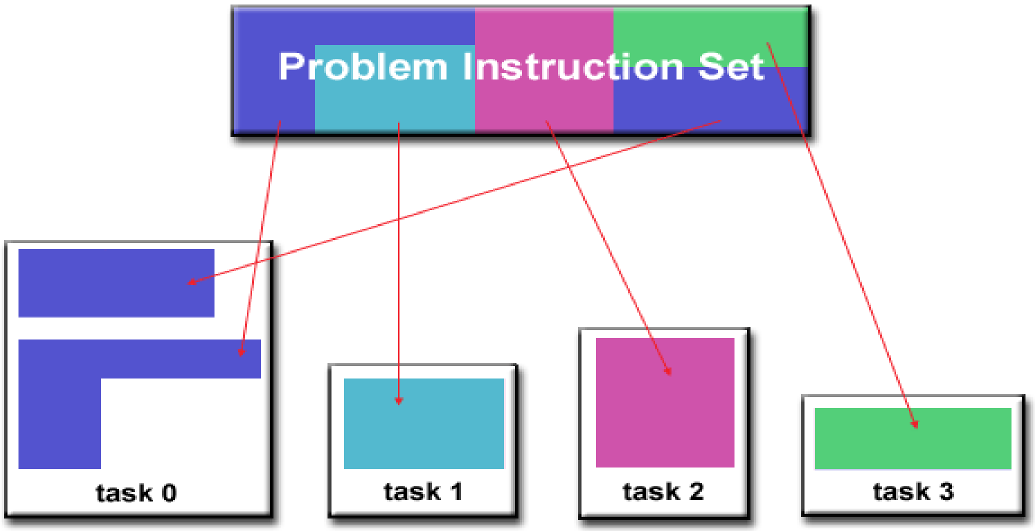
Domain Decomposition

There are different ways to partition data:



Functional Decomposition

In this approach, the focus is on the **computation that is to be performed** rather than on the data manipulated by the computation.

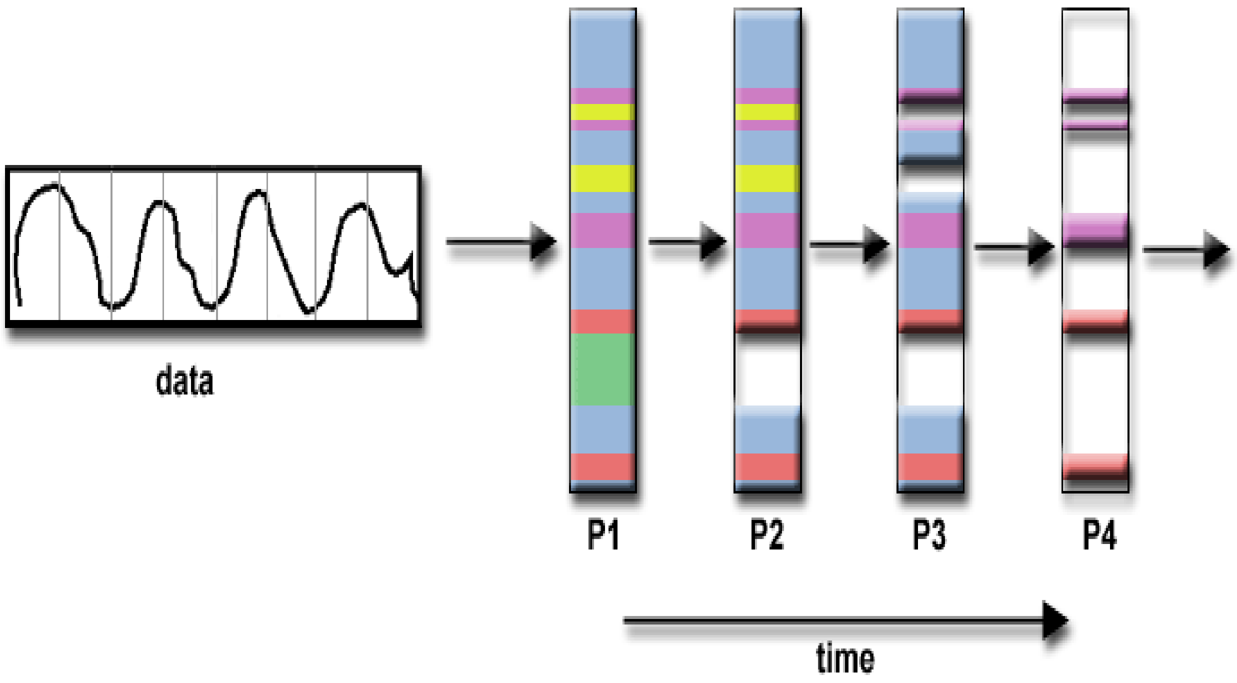


It lends itself well to **problems** that can be split into different **tasks**.

Signal Processing

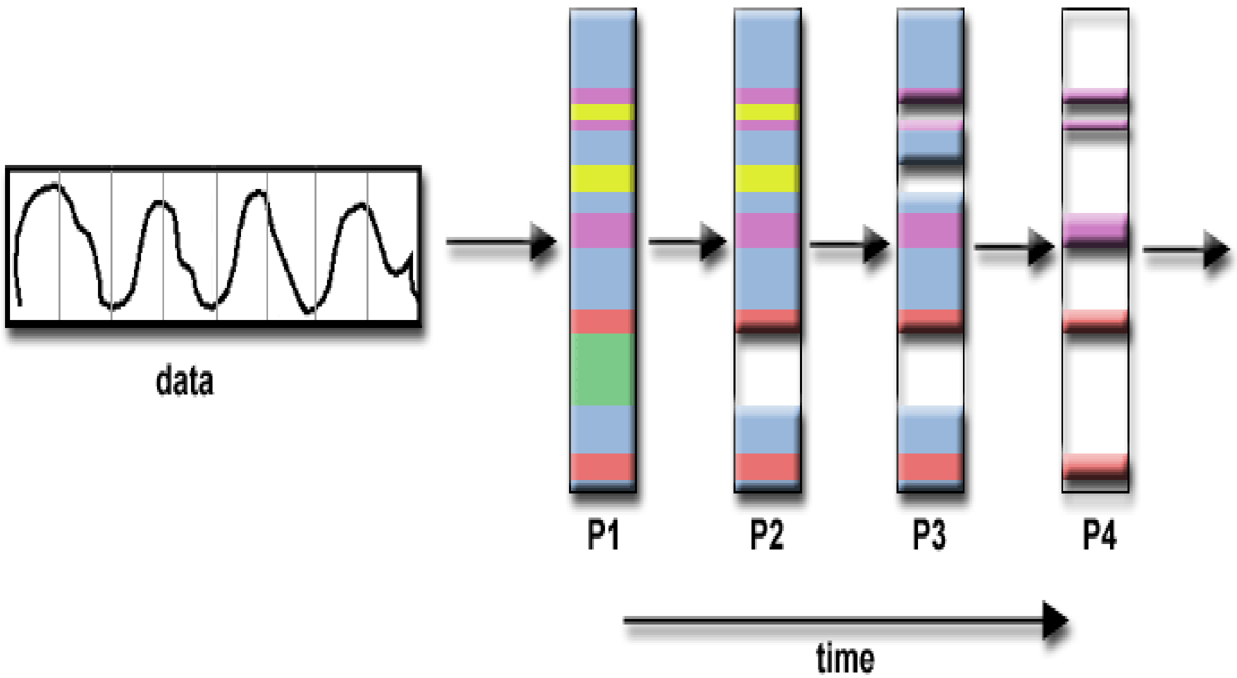
- An **audio signal data set** is passed through **four** distinct computational filters.
- Each filter is a **separate** process.
- The *first segment of data* must pass through the first filter before progressing to the second.

Signal Processing



By the time the fourth segment of data is in the first filter, all tasks are busy.

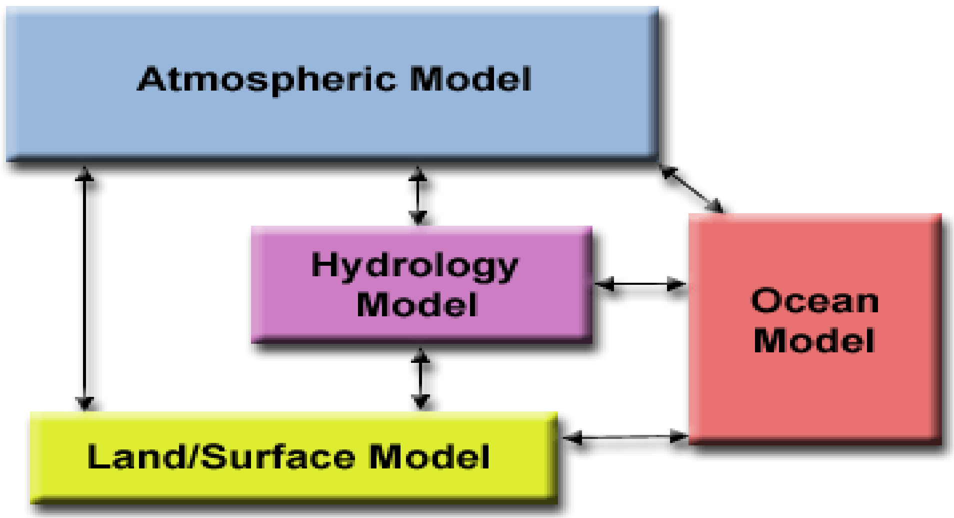
Signal Processing



By the time the fourth segment of data is in the first filter, all tasks are busy.

Climate Modeling

Each model component can be thought of as a separate task.

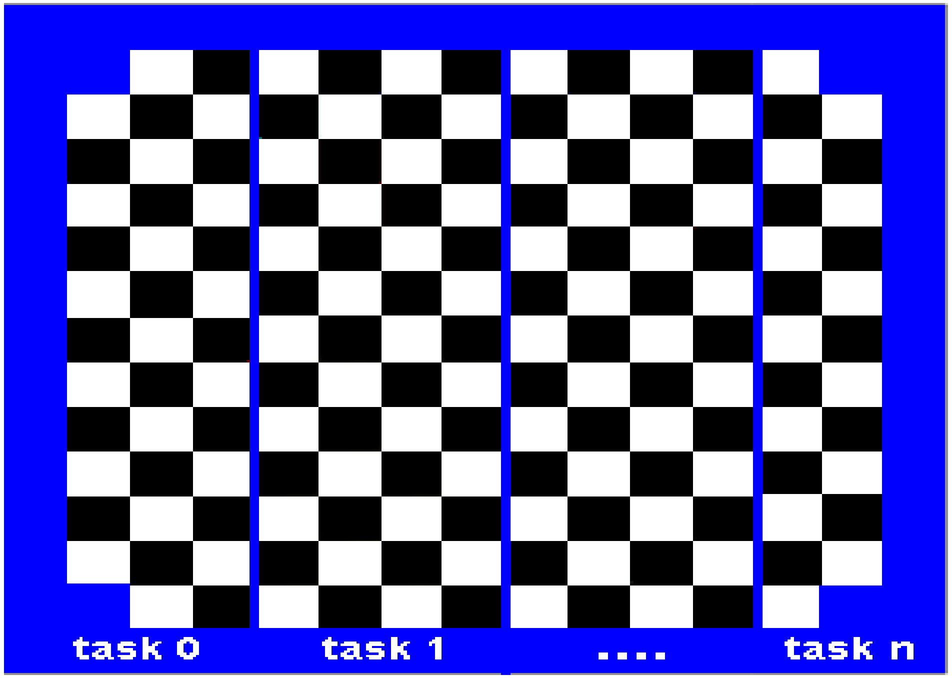


Arrow represent exchanges of data between components during computation.

Communications

You DON´T need communications:

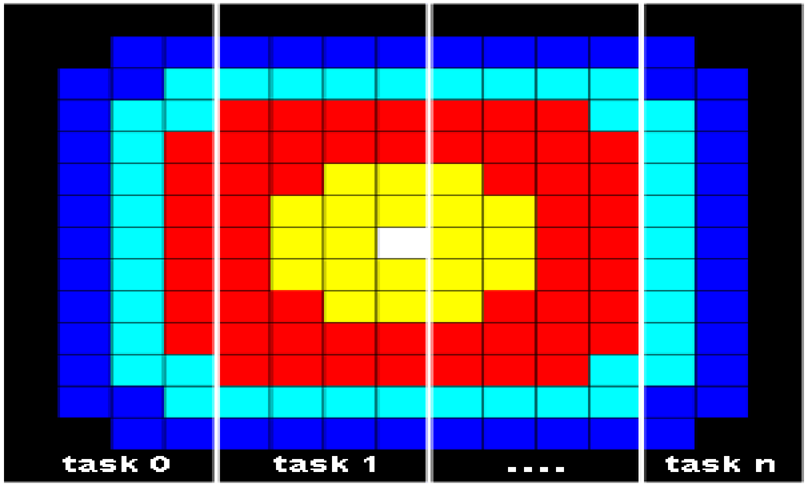
Some types of problems can be decomposed and executed in parallel.



Communications

You DO need communications:

Most parallel apps are **not** quite so simple, and do require tasks to share data with each other.



Factors to consider:

Communication overhead

- Inner-task **communication** virtually always implies **overhead**.
- Communications frequently require some type of **synchronization** between tasks → waiting
- Competing communication traffic can **sature** available network bandwidth.

Latency vs. Bandwidth

Latency → Time takes A to B.

Bandwidth → Amount of data that can be communicated per unit of time.

Sending **small messages** can cause latency to dominate communication overheads.

Visibility of communications

Message Passing Model → communications are explicit and under control the programmer.

Data Parallel Model → communications are transparent to the programmer.

Thus, the programmer may not be able **to know exactly** how inter-task communications are being accomplished.

Synchronous vs. asynchronous

Synchronous → handshaking protocol to communicate.

Synchronous communication are **blocking** → wait

Asynchronous communications allow **tasks** to **transfer data independently** from one another.

Asynchronous communications are **non-blocking**.

Interleaving **computation** with **communicion** → greatest benefit for using asynchronous comm.