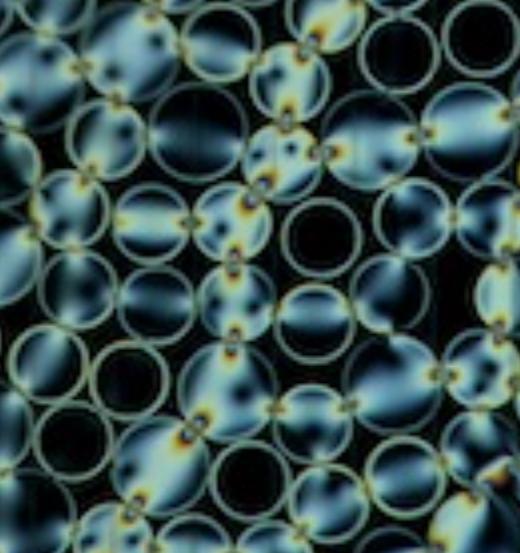
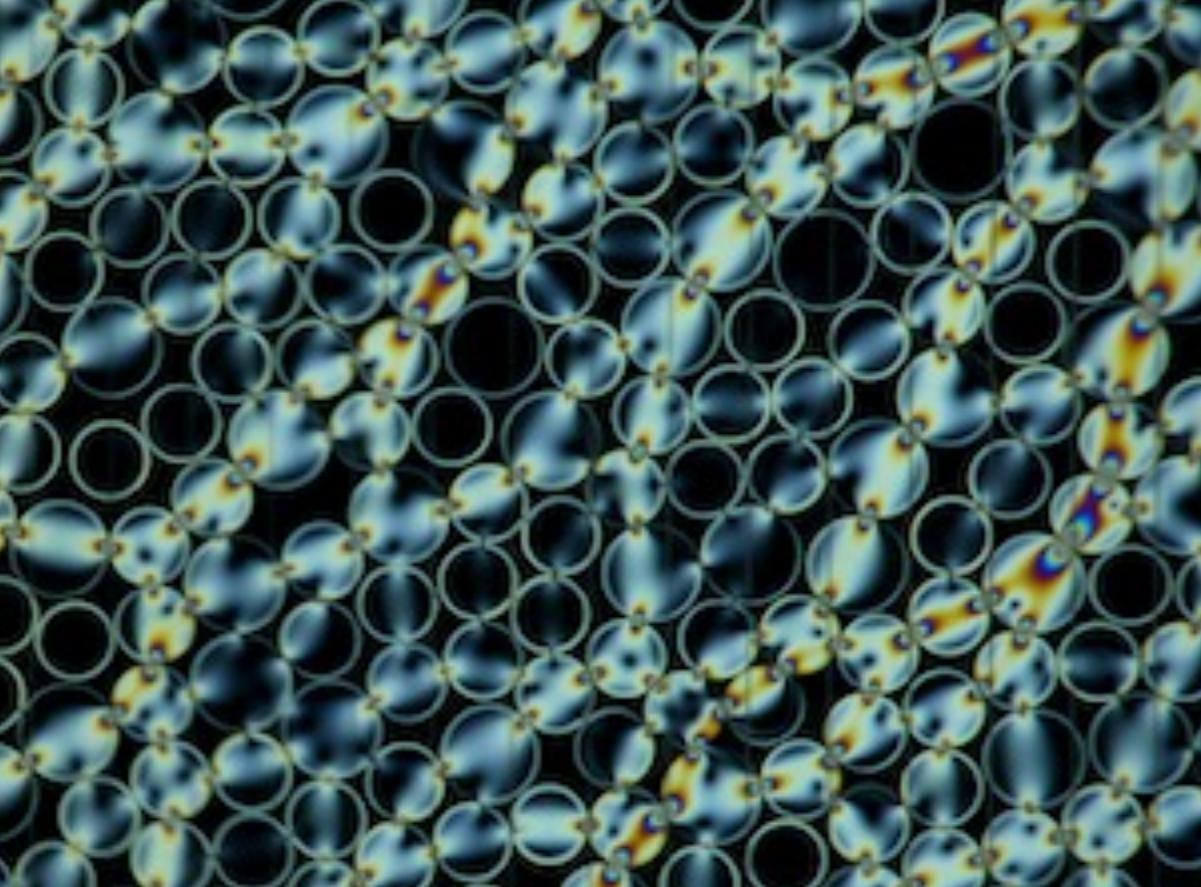


Thinking Parallel 101

Ivan Girotto
[\(igirotto@ictp.it\)](mailto:(igirotto@ictp.it))

June 2025

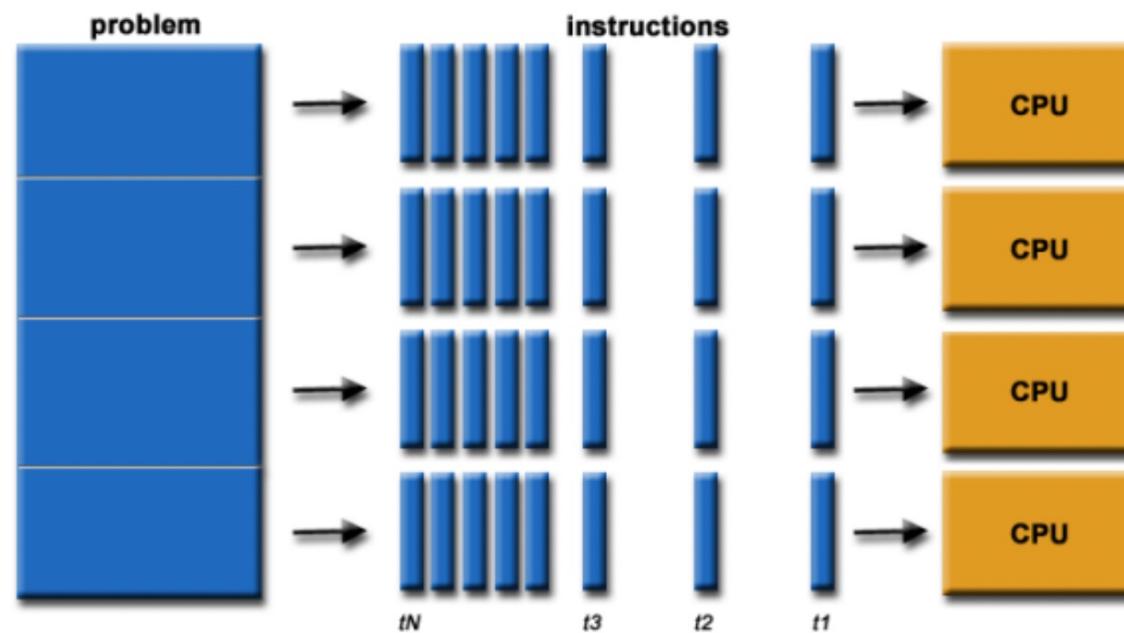


The Abdus Salam
International Centre
for Theoretical Physics



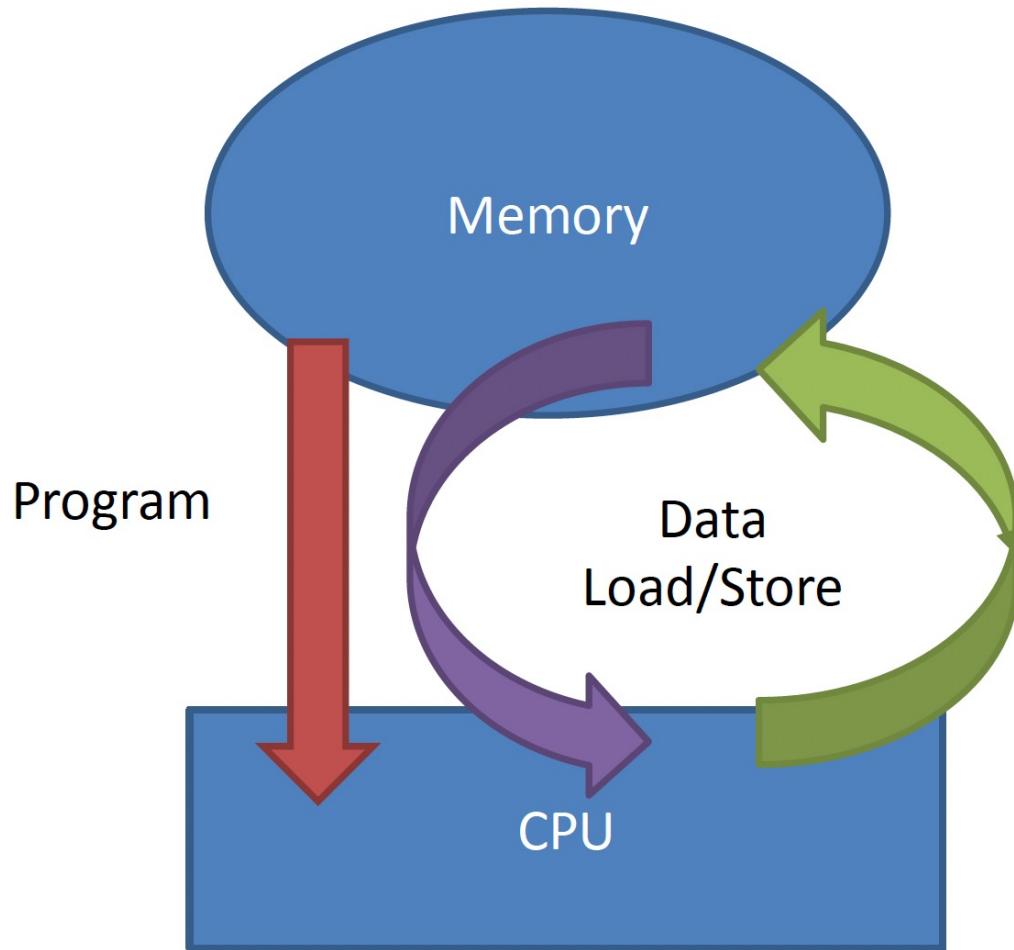
Concurrency

The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently

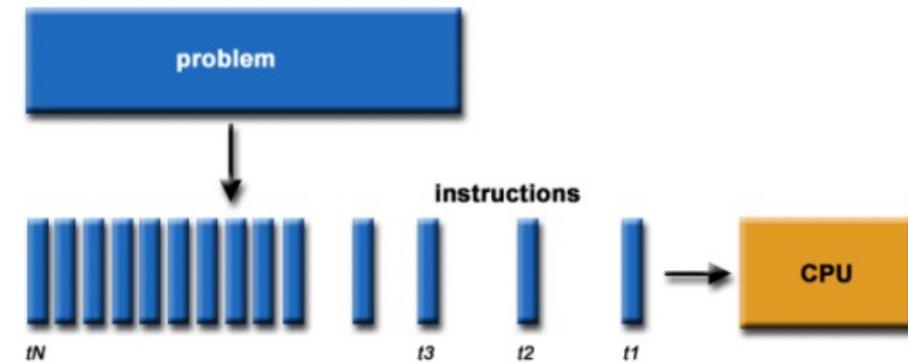


- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control / coordination mechanism is employed

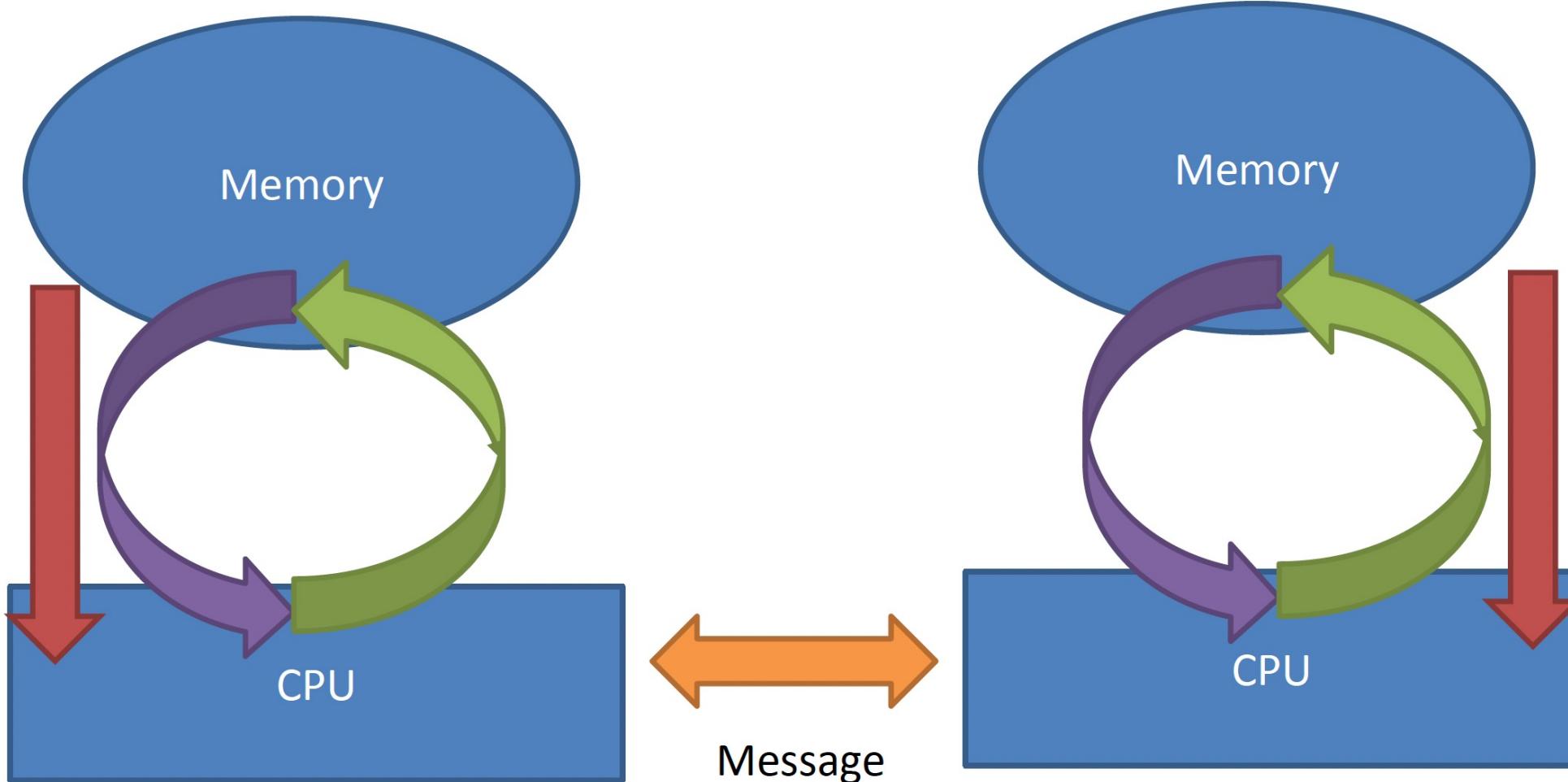
Serial Programming



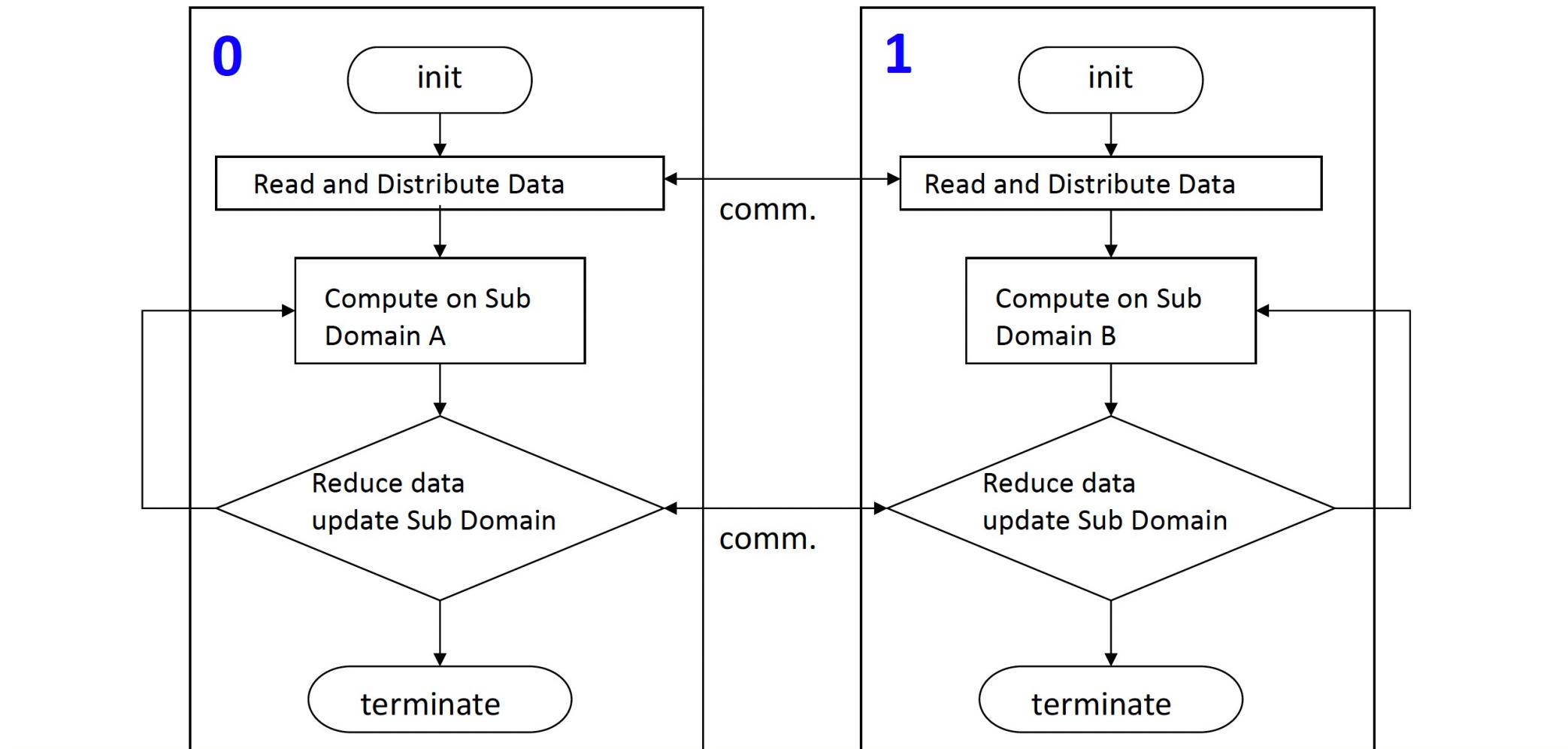
A problem is broken into a discrete series of instructions.
Instructions are executed one after another.
Only one instruction may execute at any moment in time.



Parallel Programming



What is a Parallel Program



Fundamental Steps of Parallel Design

- Identify portions of the work that can be performed concurrently
- Mapping the concurrent pieces of work onto multiple processes running in parallel
- Distributing the input, output and intermediate data associated within the program
- Managing accesses to data shared by multiple processors
- Synchronizing the processors at various stages of the parallel program execution

Type of Parallelism

- **Functional (or task) parallelism:** different people are performing different task at the same time
 - **Data Parallelism:** different people are performing the same task, but on different equivalent and independent objects



Process Interactions

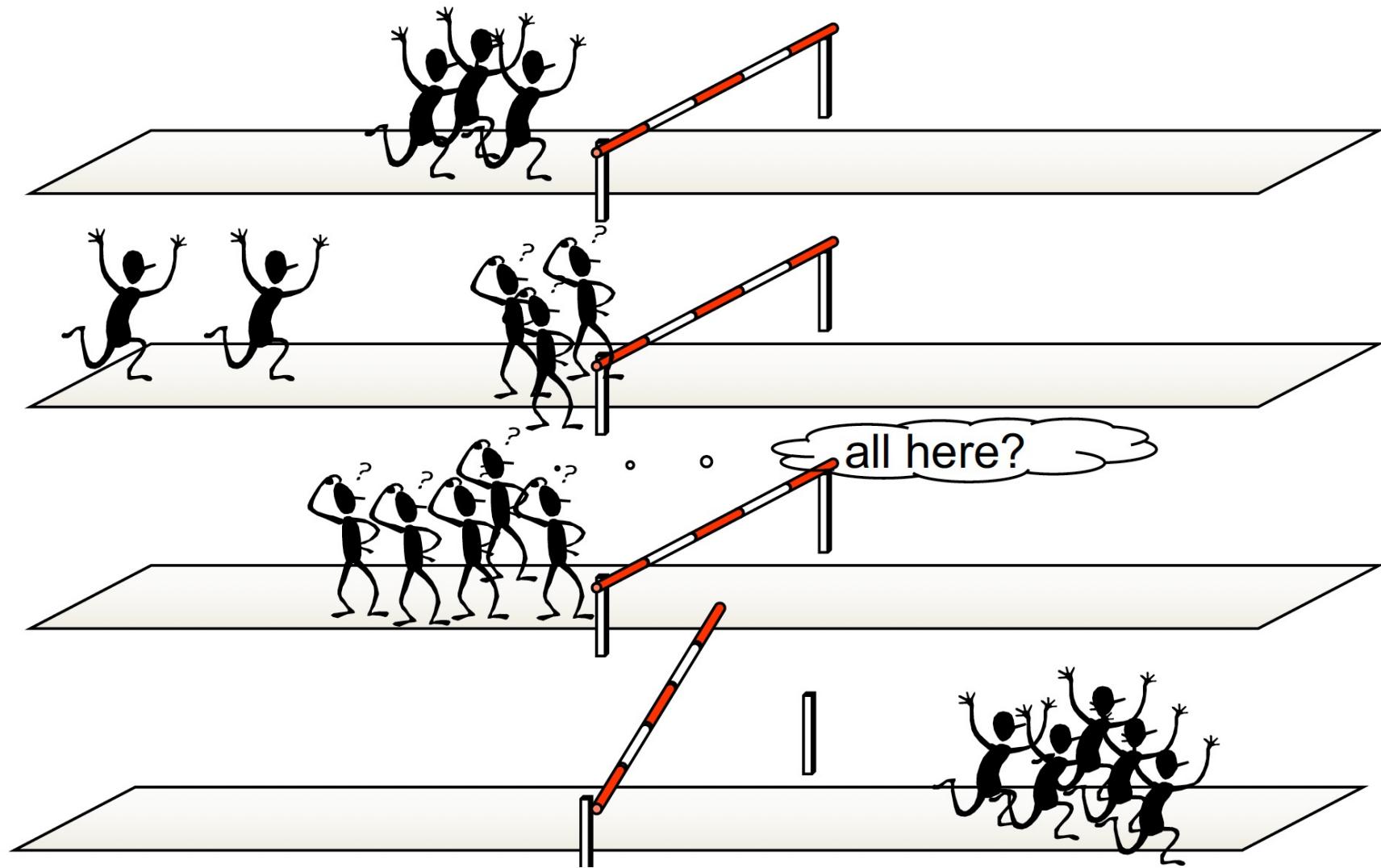
- The effective speed-up obtained by the parallelization depend by the amount of overhead we introduce making the algorithm parallel
- There are mainly two key sources of overhead:
 1. Time spent in inter-process interactions (**communication**)
 2. Time some process may spent being idle (**synchronization**)



Load Balancing

- Equally divide the work among the available resource: processors, memory, network bandwidth, I/O, ...
- This is usually a simple task for the problem decomposition model
- It is a difficult task for the functional decomposition model

Effect of Load Unbalancing



Minimizing Communication

- When possible reduce the communication events:
 - group lots of small communications into large one
 - eliminate synchronizations as much as possible.
Each synchronization level off the performance to that of the slowest process

Static Data Partitioning

The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.

row-wise distribution

P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7

column-wise distribution

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7

Distributed Data Vs Replicated Data

- Replicated data distribution is useful if it helps to reduce the communication among process at the cost of bounding scalability
- Distributed data is the ideal data distribution but not always applicable for all data-sets
- Usually complex application are a mix of those techniques => distribute large data sets; replicate small data

Global Vs Local Indexes

- In sequential code you always refer to global indexes
- With distributed data you must handle the distinction between global and local indexes (and possibly implementing utilities for transparent conversion)

Local Idx	<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr></table>	1	2	3	<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr></table>	1	2	3	<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr></table>	1	2	3
1	2	3										
1	2	3										
1	2	3										
Global Idx	<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr></table>	1	2	3	<table border="1"><tr><td>4</td><td>5</td><td>6</td></tr></table>	4	5	6	<table border="1"><tr><td>7</td><td>8</td><td>9</td></tr></table>	7	8	9
1	2	3										
4	5	6										
7	8	9										

Block Array Distribution Schemes

Block distribution schemes can be generalized to higher dimensions as well.

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

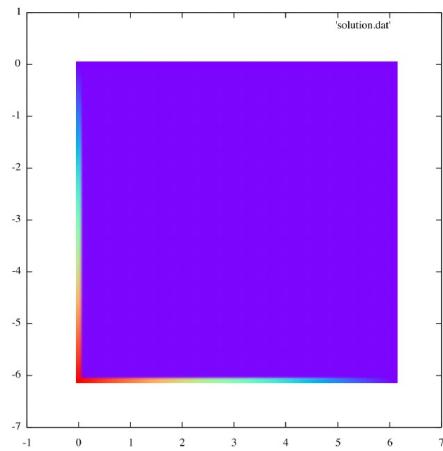
(a)

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}

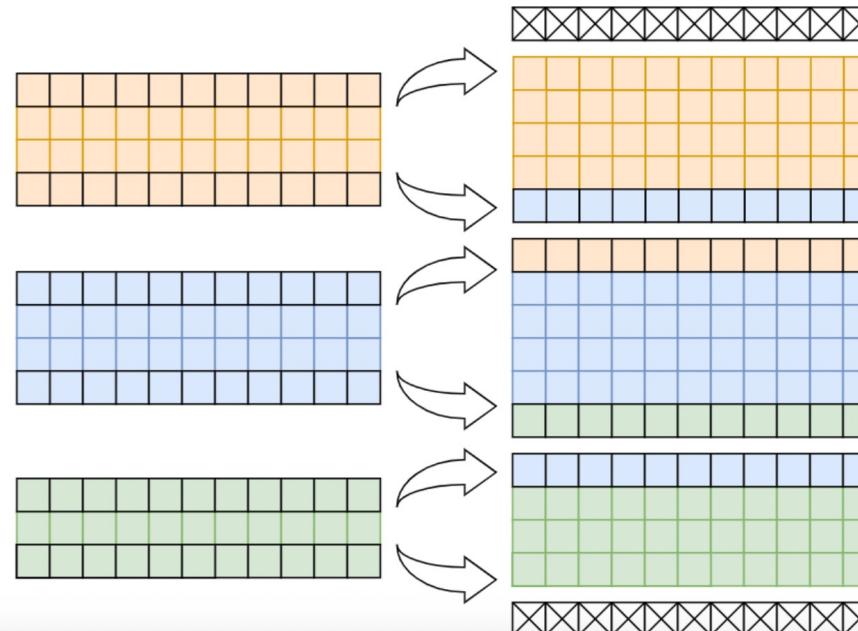
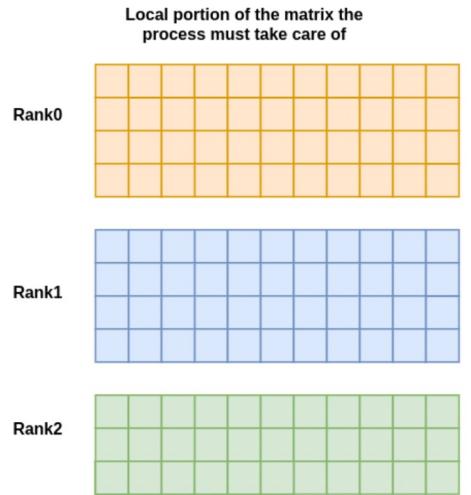
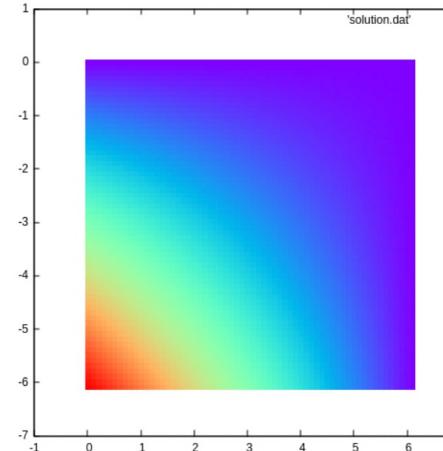
(b)

Degree to which tasks/data can be subdivided is limit to concurrency and parallel execution!!

Parallel Efficiency: Jacobi example



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

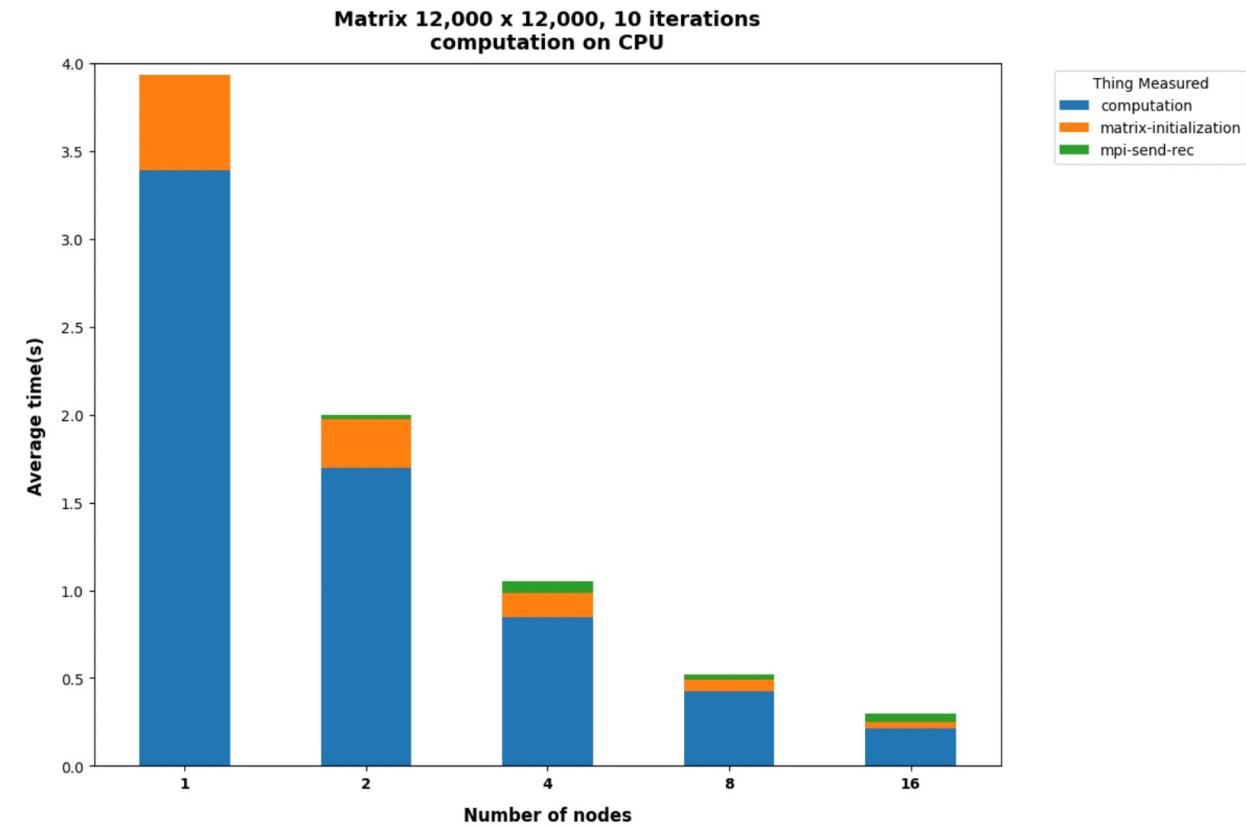
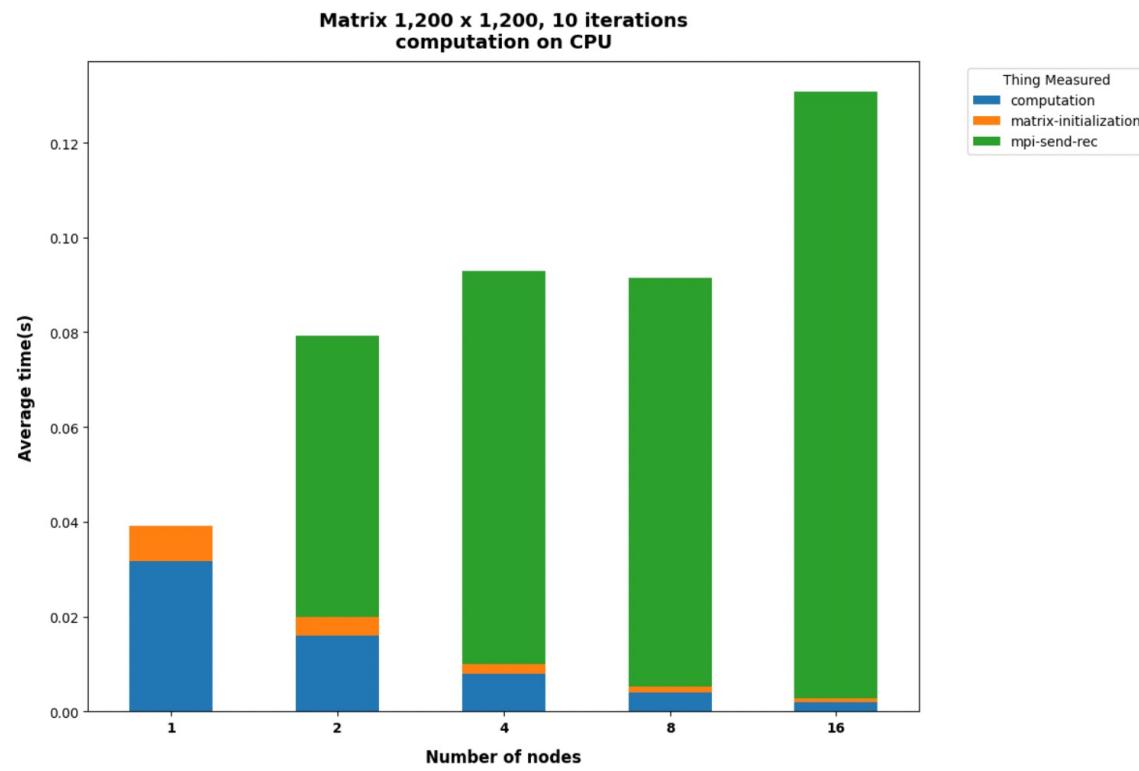


$$V_{i,j}^{new} = 0.25(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1})$$

0	0	0	0	0	0	0	0	0	0	0
10										0
20										0
30										0
40			$V_{i,j}$	$V_{i+1,j}$	$V_{i,j+1}$	$V_{i,j-1}$				0
50										0
60										0
70										0
80										0
90										0
100	90	80	70	60	50	40	30	20	10	0

Figure 1: A diagram of the Jacobi Relaxation for Solving the Laplace's Equation on an evenly spaced 9x9 grid with the boundary conditions outlined in the text above.

Parallel Efficiency: Jacoby example



Overlap Communication and Computation

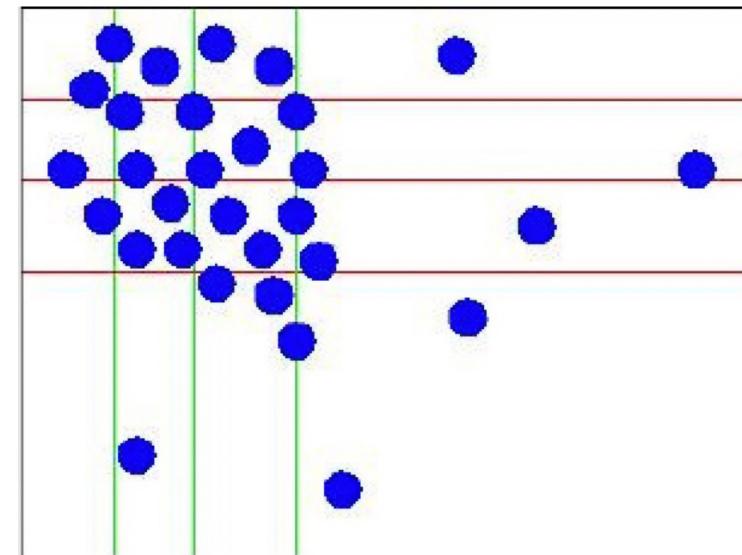
- When possible code your program in such a way that processes continue to do useful work while communicating
- This is usually a non trivial task and is afforded in the very last phase of parallelization
- If you succeed, you have done. Benefits are enormous

Granularity

- Granularity is determined by the decomposition level (number of task) on which we want divide the problem
- The degree to which task/data can be subdivided is limit to concurrency and parallel execution
- Parallelization has to become “topology aware”
 - coarse grain and fine grained parallelization has to be mapped to the topology to reduce memory and I/O contention
 - make your code modularized to enhance different levels of granularity and consequently to become more “platform adaptable”

Limitations of Parallel Computing

- Fraction of serial code limits parallel speedup
- Degree to which tasks/data can be subdivided is limit to concurrency and parallel execution
- Load imbalance:
 - parallel tasks have a different amount of work
 - CPUs are partially idle
 - redistributing work helps but has limitations
 - communication and synchronization overhead



Shared Resources

- In parallel programming, developers must manage exclusive access to shared resources
- Resources are in different forms:
 - concurrent read/write (including parallel write) to shared memory locations
 - concurrent read/write (including parallel write) to shared devices
 - a message that must be send and received

Fundamental Tools of Parallel Programming



Programming Parallel Paradigms

- Are the tools we use to express the parallelism for on a given architecture
- They differ in how programmers can manage and define key features like:
 - parallel regions
 - concurrency
 - process communication
 - synchronism



Workload Management: system level, High-throughput

Python: Ensemble simulations, workflows

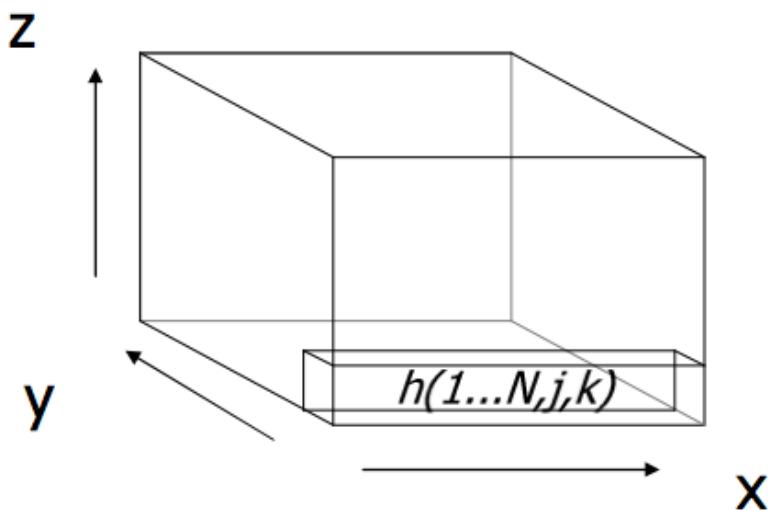
MPI: Domain partition

OpenMP: Node Level shared mem

CUDA/OpenCL/OpenAcc:
floating point accelerators

Challenge: code maintainability

Multidimensional FFT

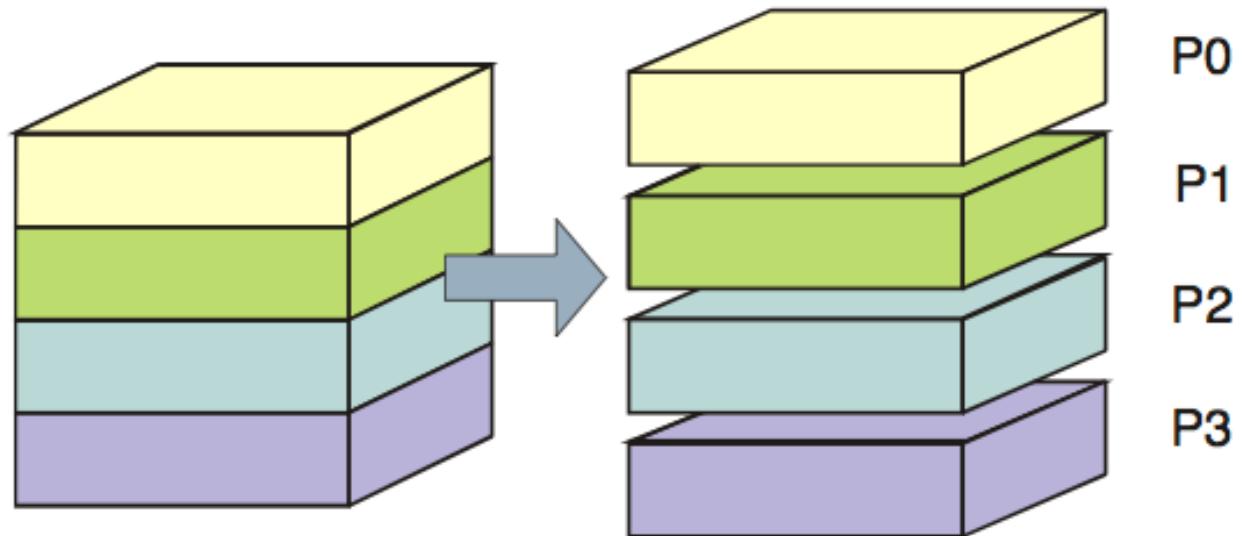


- 1) For any value of j and k
transform the column $(1\dots N, j, k)$
- 2) For any value of i and k
transform the column $(i, 1\dots N, k)$
- 3) For any value of i and j
transform the column $(i, j, 1\dots N)$

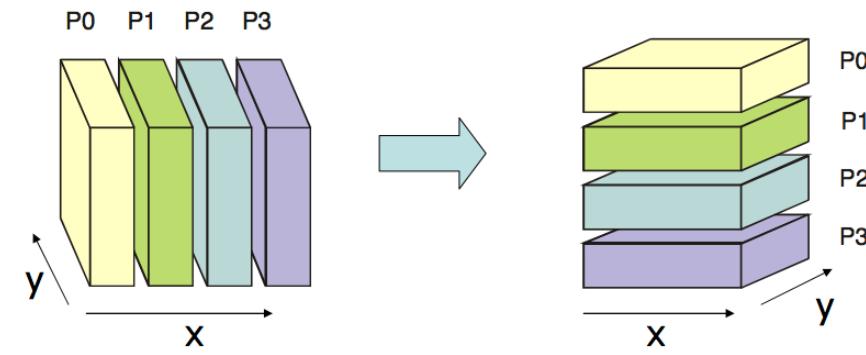
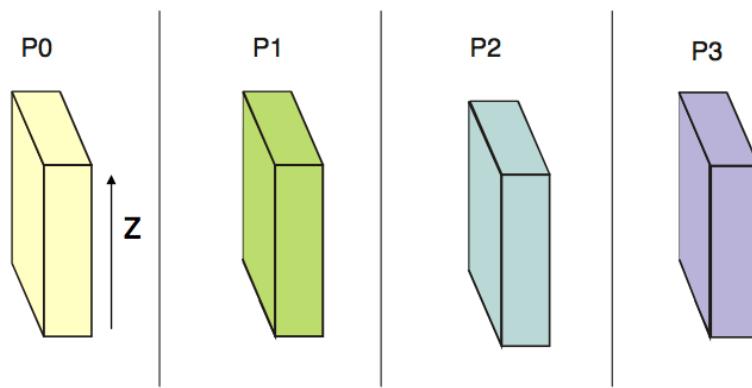
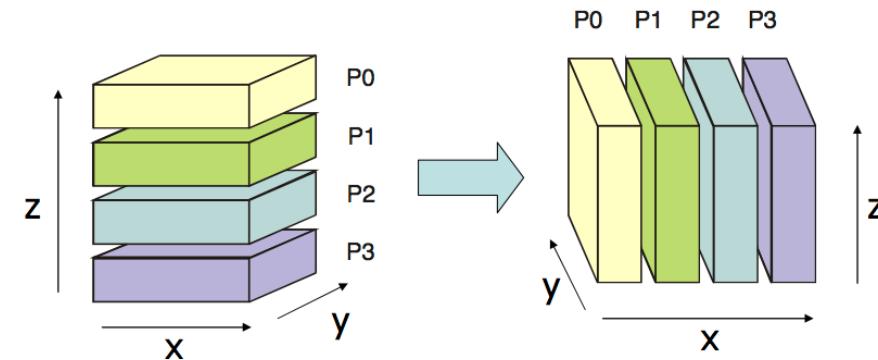
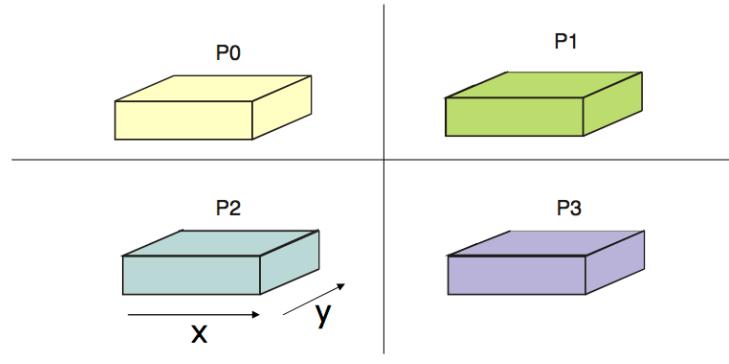
$$f(x, y, z) = \frac{1}{N_z N_y N_x} \sum_{z=0}^{N_z-1} \left(\sum_{y=0}^{N_y-1} \left(\underbrace{\sum_{x=0}^{N_x-1} F(u, v, w) e^{-2\pi i \frac{xu}{N_x}}}_{\text{DFT long x-dimension}} e^{-2\pi i \frac{yu}{N_y}} \right) e^{-2\pi i \frac{zw}{N_z}} \right)$$

$\underbrace{\hspace{10em}}$
DFT long y-dimension
 $\underbrace{\hspace{10em}}$
DFT long z-dimension

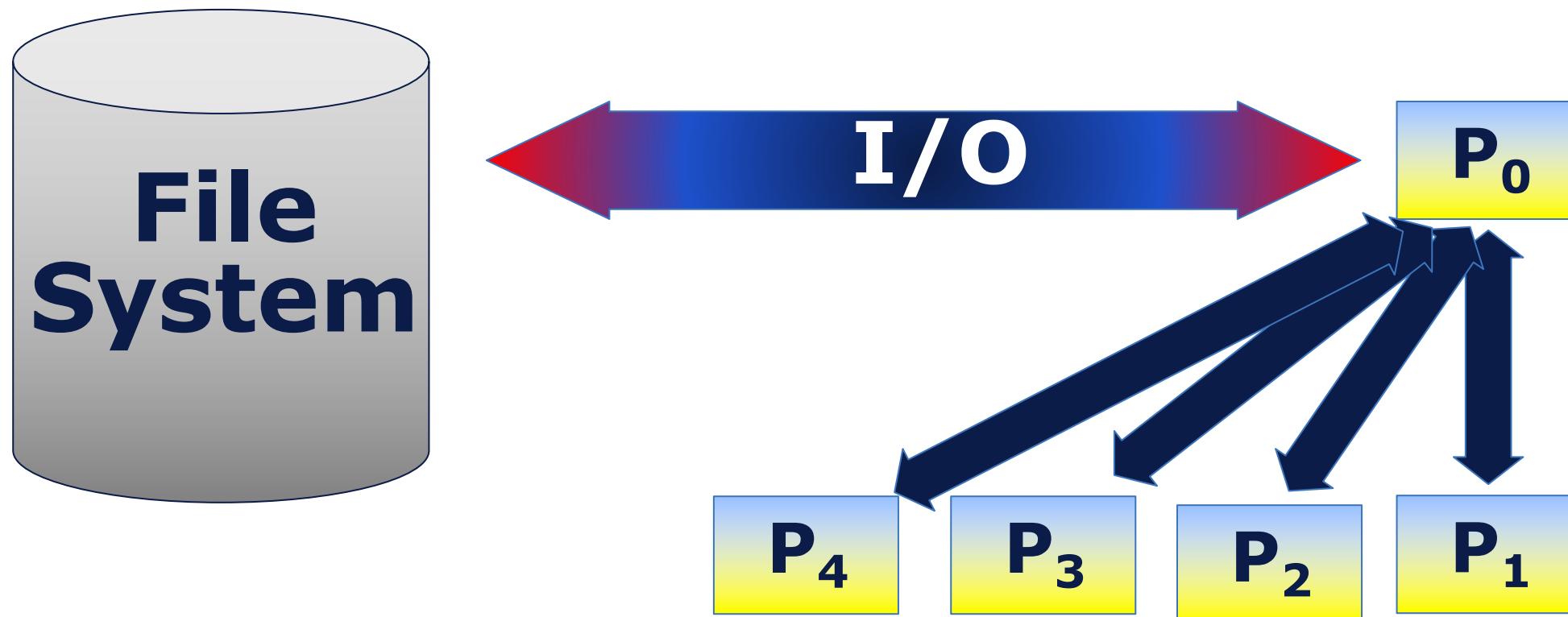
Parallel 3DFFT / 1



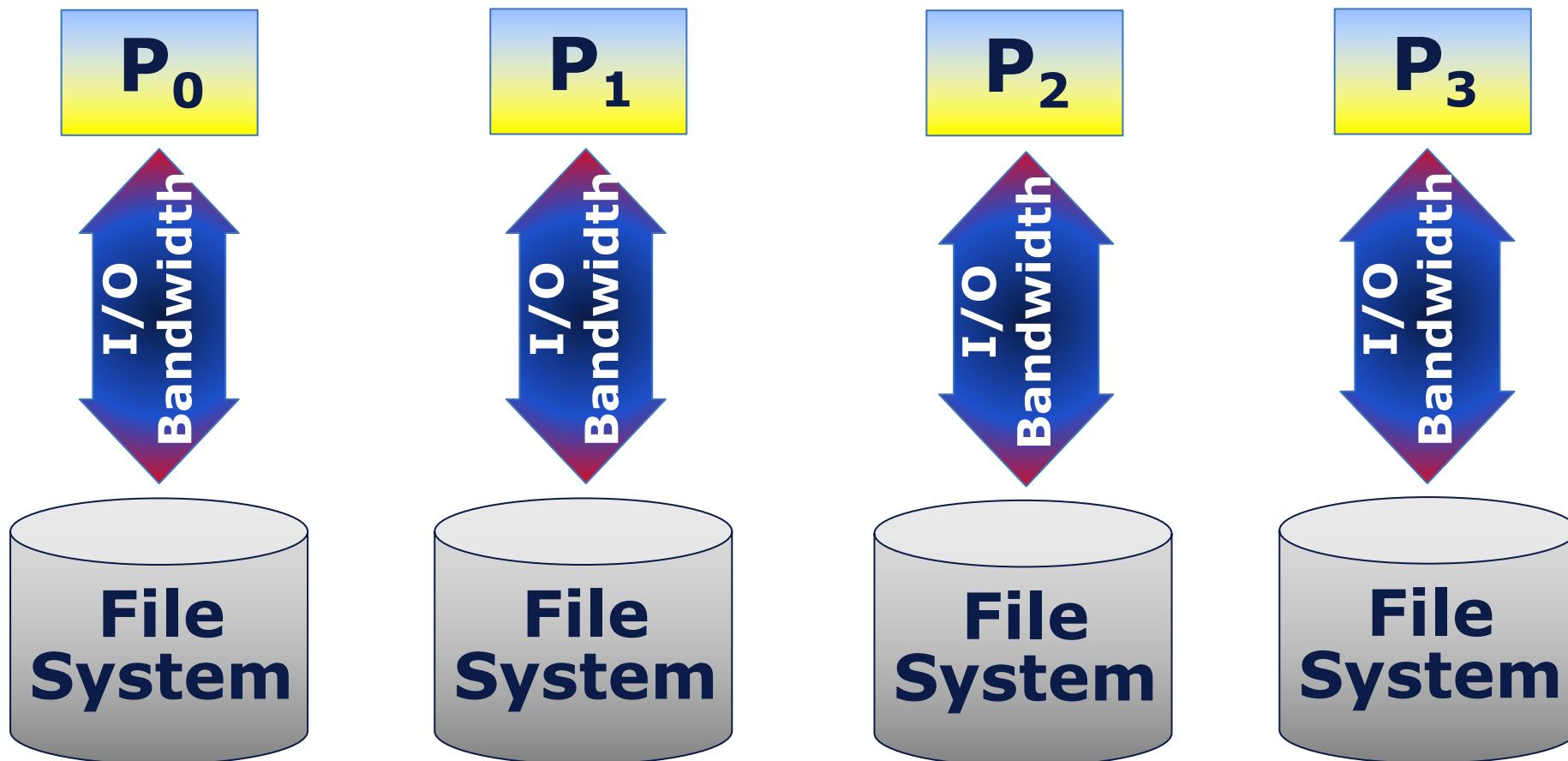
Parallel 3DFFT / 2



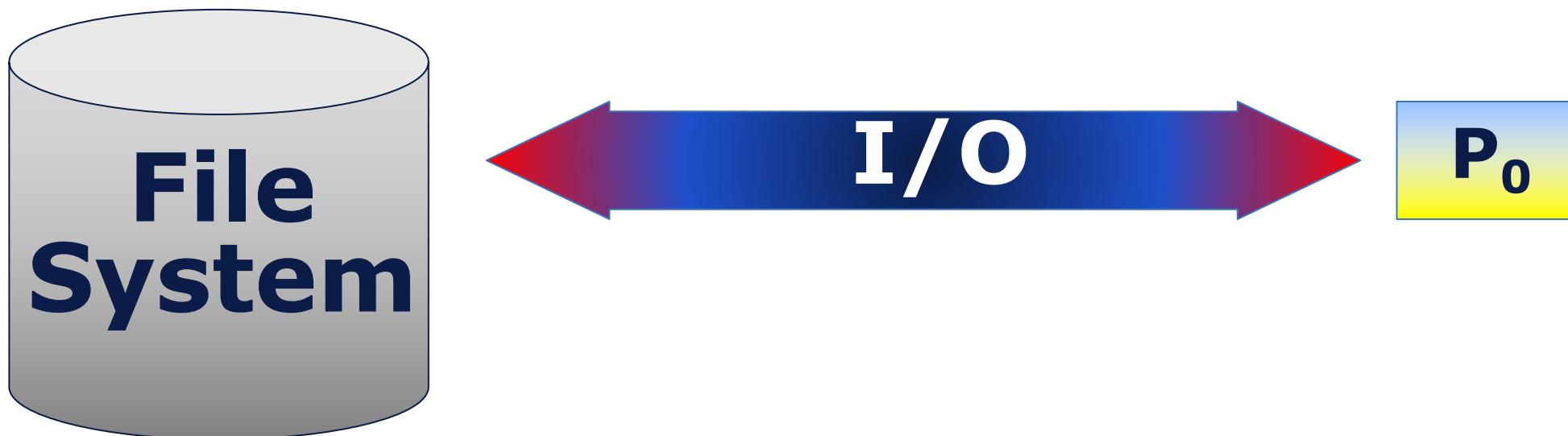
Parallel I/O



Parallel I/O



Serial I/O



Parallel I/O

P₀



P₁



P₂

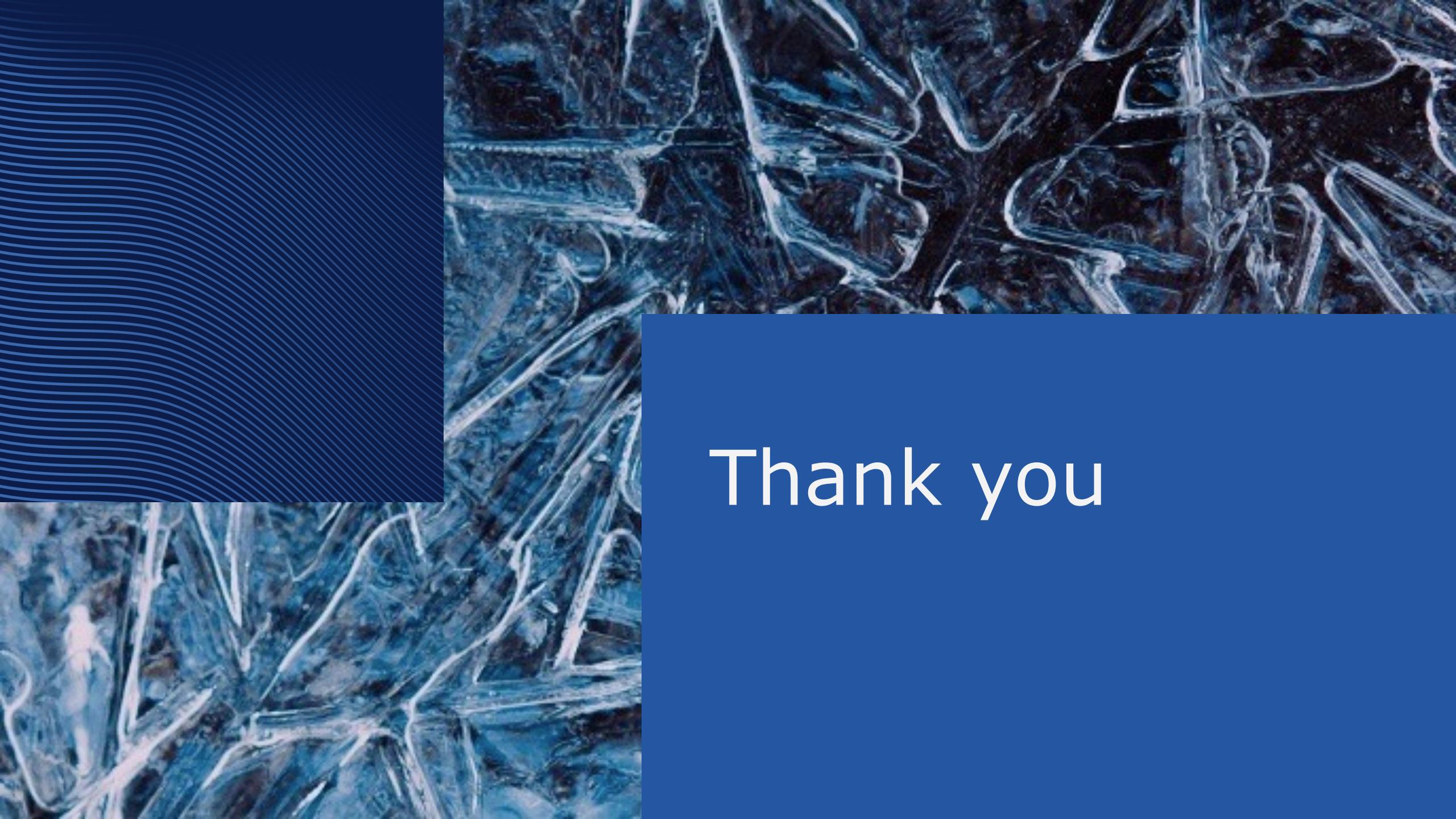


P₃



**MPI I/O & Parallel I/O Libraries
(Hdf5, Netcdf, etc...)**





Thank you