IN3200/IN4200: Chapter 5
Basics of parallelization

Sections 5.3.5, 5.3.6, 5.3.7, 5.3.8 are not required

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Overview

High-performance computing = efficient serial computing + effective parallel processing $\rightarrow$ needs parallel programming

But before actually engaging in parallel programming, it is vital to know some fundamental things about **parallelization**:

- The most common strategies for parallelization
- Simple theoretical insights into the factors that can hamper parallel performance (to be covered in part 2)

Why parallelize?

- We want to solve the existing problems faster, but the speed of a single CPU core has "saturated".
- We want to solve larger problems, but the main memory available on a single system is not large enough.

So, we need to **identify parallelism** in a given computational problem, so that parallel programming can produce a parallel implementation that can efficiently use many processor cores, on a shared- or distributed-memory system.

Identifying parallelism

The first step is to identify the parallelism inherent in the algorithm at hand—how can *multiple* compute elements collaborate to solve the computational problem?

Different variants of parallelism induce different methods of parallelization.

Data parallelism

Most problems in scientific computing involve processing of large quantities of data stored on a computer.

If this can be performed in parallel by multiple processors concurrently working on different parts of the data—**data parallelism**.

This is the dominant parallelization concept in scientific computing, also goes under the name *SPMD* (single program multiple data).

Example: medium-grained loop parallelism

Processing of array data by loops or loop nests is a central component in most scientific codes.

When computations performed on the individual array elements are independent of each other—typical candidates for parallel execution by multiple "workers" with help of shared memory. (Also possible on distributed-memory systems after appropriate data partitioning.)

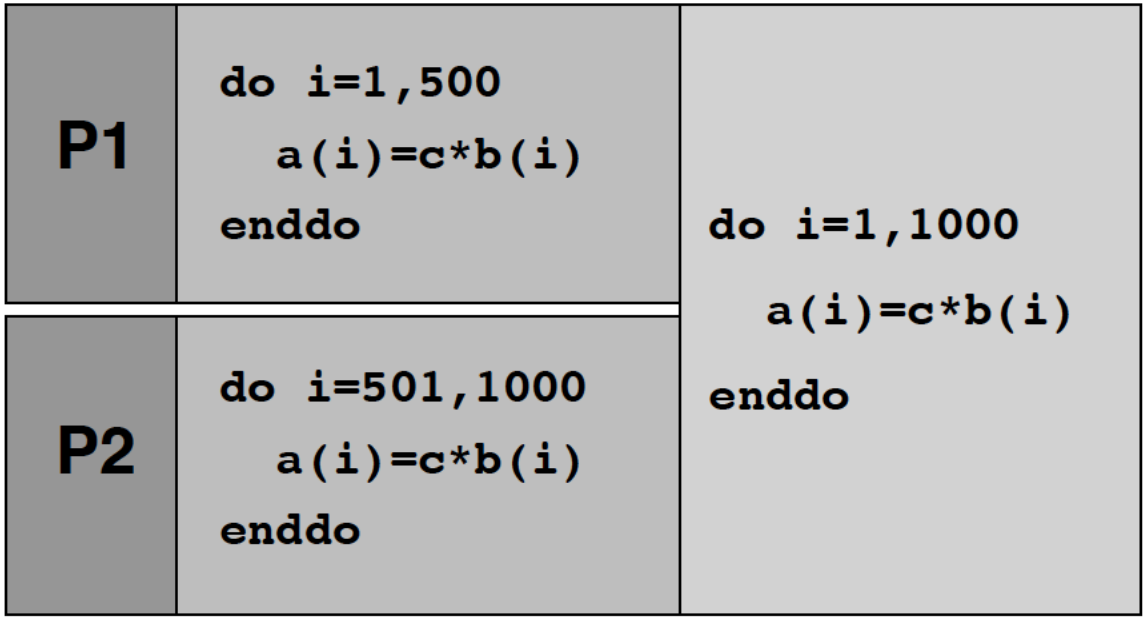


Figure 5.1: An example for medium-grained parallelism: The iterations of a loop are distributed to two processors P1 and P2 (in shared memory) for concurrent execution.

Questions to ponder...

Can the parallelization approach shown on the previous slide work on the following two examples? (why? or why not?)

```
for (i=1; i<n; i++)
  a[i] = c*a[i-1];
```

```
for (i=0; i<n-1; i++)
  a[i] = c*a[i+1];
```

More examples of data parallelism (matrix-matrix multiplication)

Multiplying two matrices A and B to yield matrix C. The output matrix C can for example be partitioned into four blocks (where each block is a sub-matrix):

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$\text{Process 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$\text{Process 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$\text{Process 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

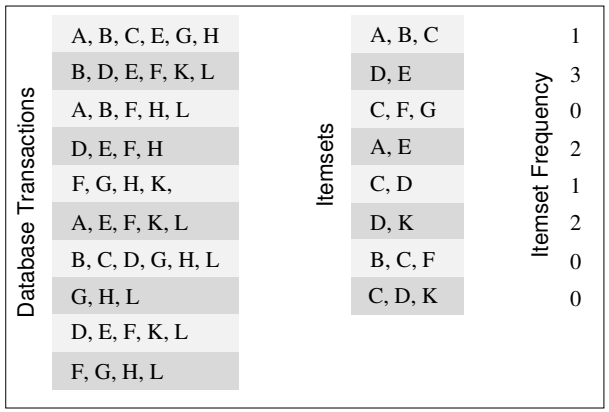
$$\text{Process 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

(Question to ponder: What to do if you have more than 4 CPUs?)

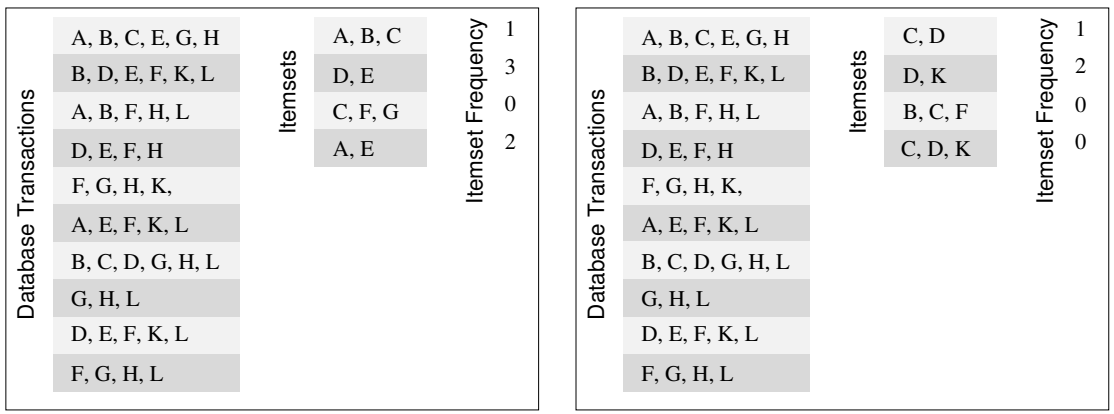
More examples of data parallelism (counting occurrences)

Counting the occurrences of given itemsets in a database of transactions. For example, the output (itemset frequencies) can be partitioned among processes.

(a) Transactions (input), itemsets (input), and frequencies (output)



(b) Partitioning the frequencies (and itemsets) among the tasks



A. Grama, A. Gupta, G. Karypis, and V. Kumar: "Introduction to Parallel Computing", Addison Wesley, 2003

More examples of data parallelism (counting occurrences)

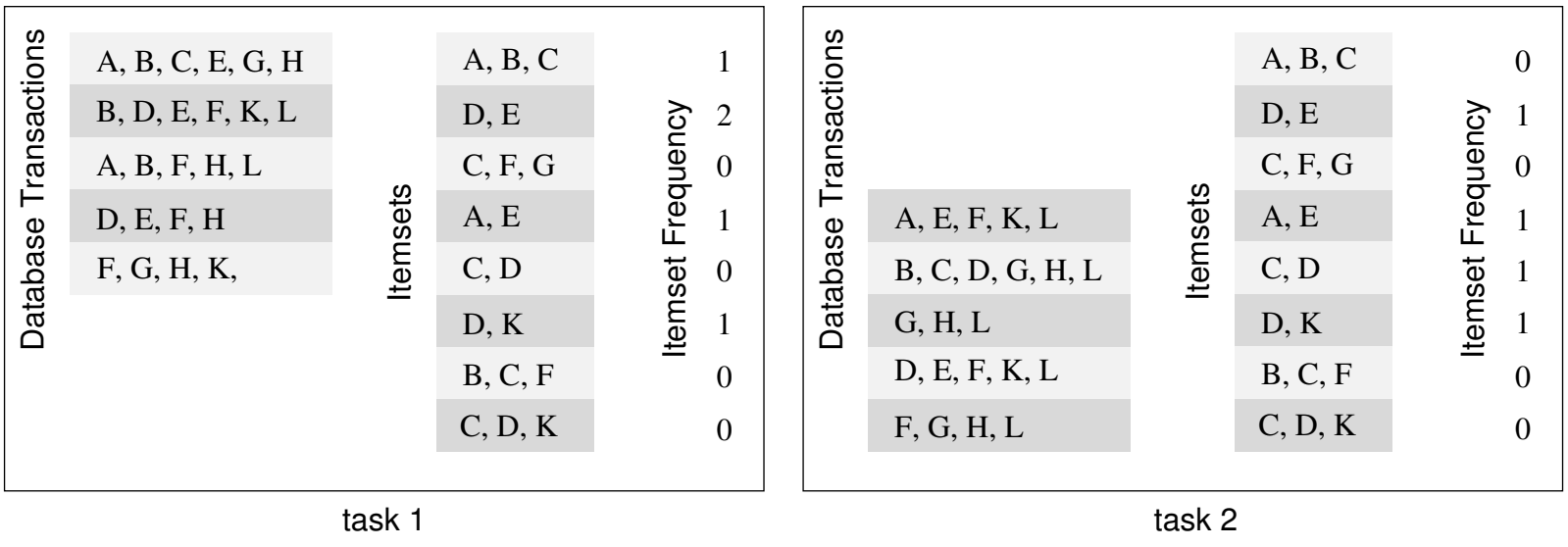
We have the following observations:

- If the database is shared (on a shared-memory system) or replicated across the processes (on a distributed-memory system), each process can be independently accomplished (no need for communication).
- If the database is partitioned across processes as well (for best use of distributed memory), each process can first compute partial counts. These counts then have to be aggregated (by communication).

More examples of data parallelism (counting occurrences)

Input Data Partitioning: For the database counting example, we can choose to partition the input (i.e., the transaction set).

Partitioning the transactions among the tasks



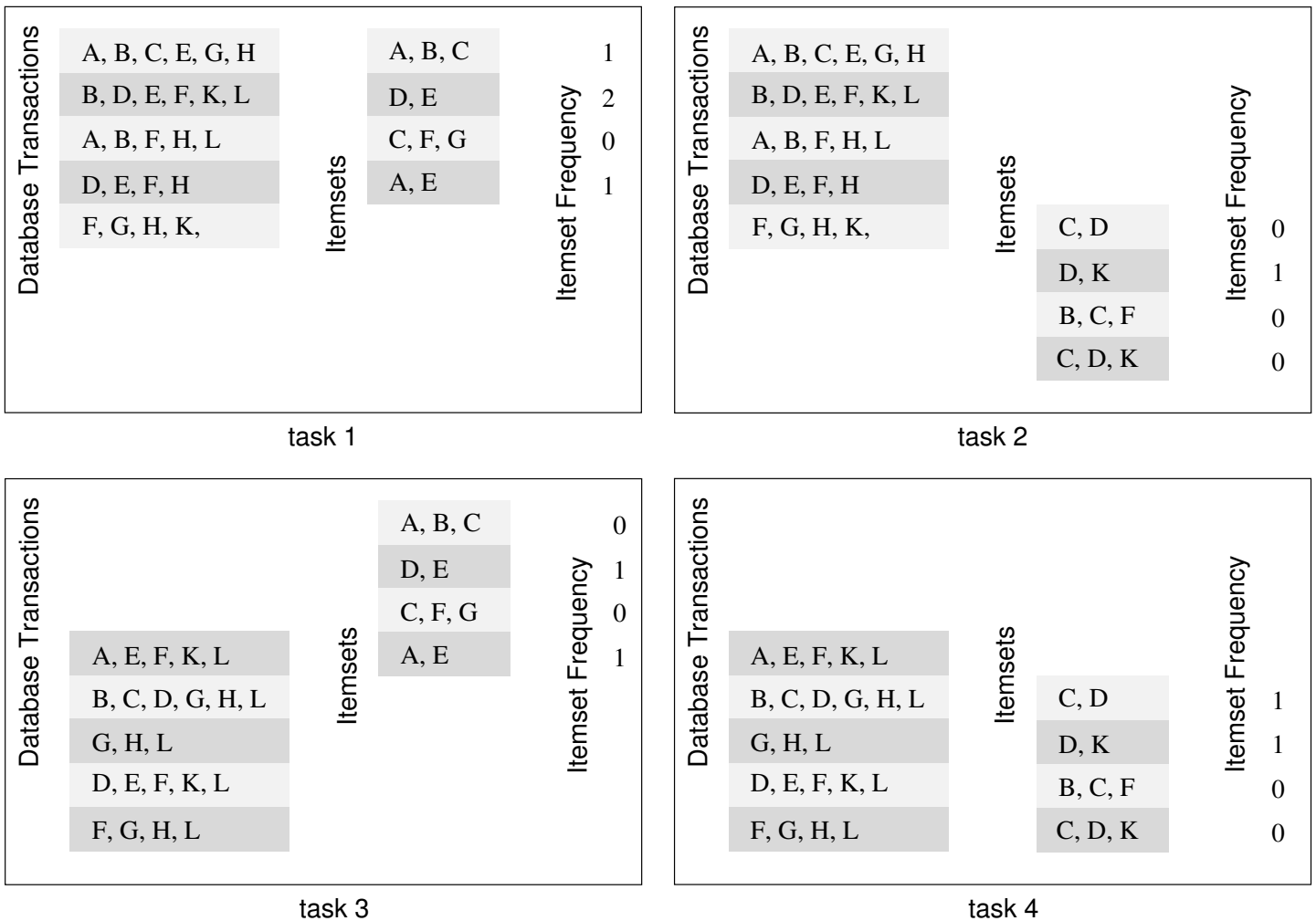
The two partial counting results of itemset frequency need to be aggregated.

A. Grama, A. Gupta, G. Karypis, and V. Kumar: "Introduction to Parallel Computing", Addison Wesley, 2003

More examples of data parallelism (counting occurrences)

Input and Output Data Partitioning (aggregation needed afterwards):

Partitioning both transactions and frequencies among the tasks

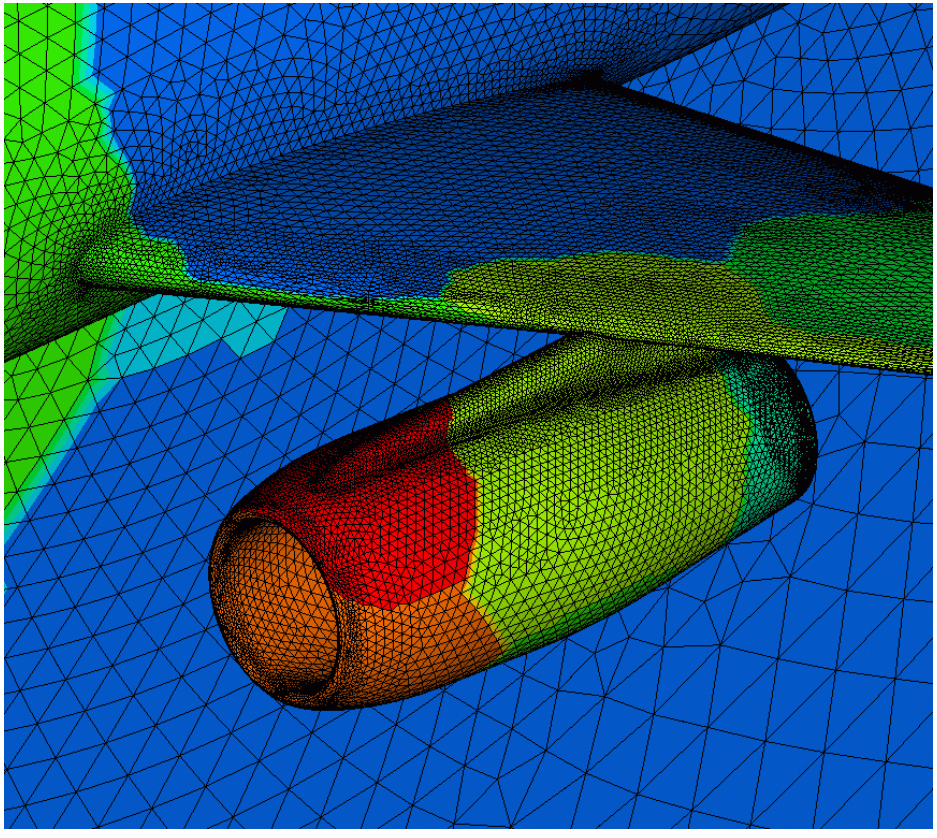


A. Grama, A. Gupta, G. Karypis, and V. Kumar: "Introduction to Parallel Computing", Addison Wesley, 2003

Example: Coarse-grained parallelism by domain decomposition

In case of a computational domain represented as a grid, a straightforward way to distribute the computation among "workers" is to assign a part of the grid to each worker—*domain decomposition*.

Works on both shared-memory and distributed-memory computers.



Recall the "2D Jacobi example"

Modified from Chapter 3.3:

```
for (it=0; it<itmax; it++) {
  for (k=1; k<n-1; k++)
    for (i=1; i<n-1; i++)
      phi_new[k][i] = (phi[k-1][i]+phi[k][i-1]
                       +phi[k][i+1]+phi[k+1][i])*0.25;
  /* pointer swapping */
  temp_ptr = phi_new;
  phi_new = phi;
  phi = temp_ptr;
}
```

Note: `phi_new` and `phi` are 2D arrays (row-major) of dimension $n \times n$

Parallelizing the "2D Jacobi example" on shared memory

Note: On the same time level (`it` iteration), computing the different `phi_new[k][i]` values is independently of each other.

On a shared-memory system, we can let multiple processes divide the `i`-indexed `for` loop, or divide the `k`-indexed `for` loop.

Question to ponder: Do we need to coordinate the different processes on a shared-memory system?

Domain decomposition & distributed memory

If domain decomposition is to be implemented for distributed memory, grid & data are **explicitly partitioned**. Updating one subdomain's boundary points requires data from one or more adjacent subdomains.

Explicit communication is thus needed, together with *halo* or *ghost* layers in the data structure.

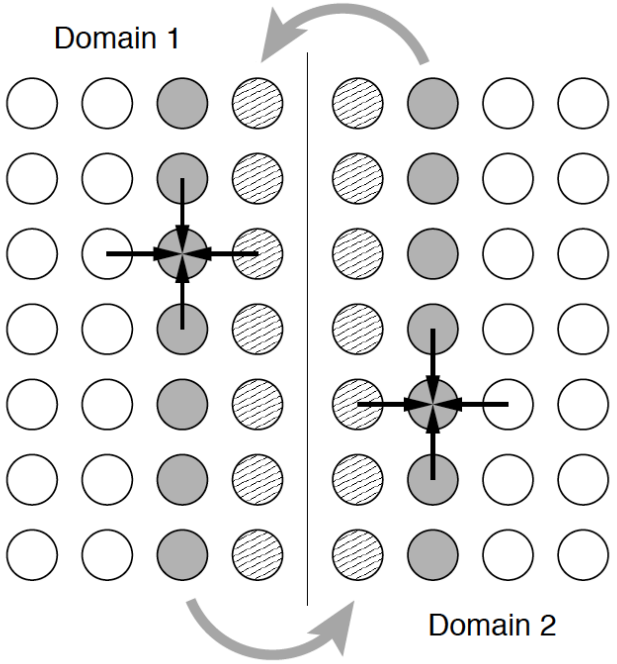


Figure 5.2: Using halo ("ghost") layers for communication across domain boundaries in a distributed-memory parallel Jacobi solver. After the local updates in each domain, the boundary layers (shaded) are copied to the halo of the neighboring domain (hatched).

More about domain decomposition

Unless the computational grid is regular, domain decomposition can be a non-trivial task:

- Load balance—all "workers" get roughly the same amount of computation;
- Communication overhead—must be kept low;

Impact of decomposition on communication overhead

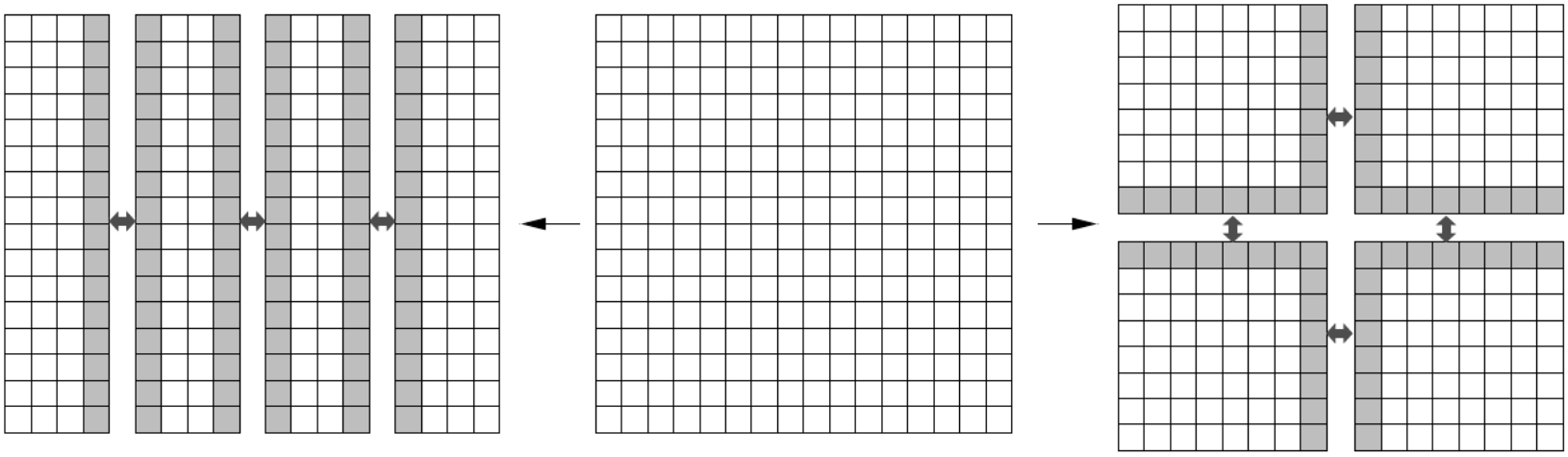
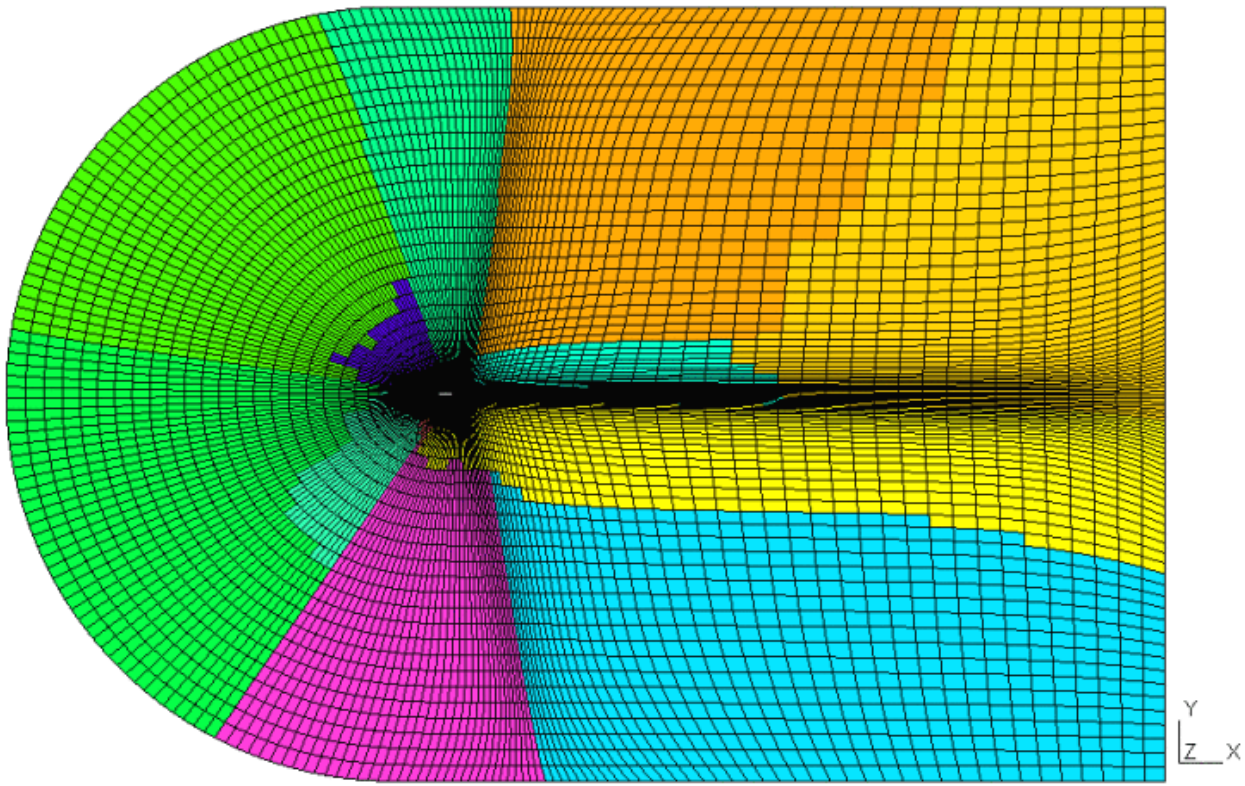


Figure 5.3: Domain decomposition of a two-dimensional Jacobi solver, which requires next-neighbor interactions. Cutting into stripes (left) is simple but incurs more communication than optimal decomposition (right). Shaded cells participate in network communication.

Example of general decomposition



https://nutscfd.wordpress.com/2017/03/06/mesh-partitioning-using-parmetis/

Functional parallelism

When the solution of a "big" problem can be split into disparate subtasks, which work together by data exchange and synchronization. The subtasks execute completely different code on different data items, so **functional parallelism** becomes an option.

Functional parallelism is also called *MPMD* (multiple program multiple data). Each subtask may contain data parallelism and be further parallelized by SPMD.

- If different parts of the "big" problem have different performance properties and hardware requirements, bottlenecks and load imbalance can easily arise.
- Overlapping subtasks—in the "design" of functional parallelism—can give performance benefits.

Example: master-worker scheme

Reserving one "master" compute element for administrating the other compute elements (as "workers").

- From a pool of subtasks, the master dynamically assigns the workers with computational work.
- The master is also responsible for collecting results from the workers.

A drawback of the master-worker scheme is the potential communication and performance bottleneck that may appear with a single master that administrates a large number of workers.

Example: functional decomposition

Multiphysics simulations are possible candidates for parallelization by functional decomposition (combined with data parallelism).

For instance, the air flow around a racing car can be simulated using a parallel CFD (computational fluid dynamics) code. On the other hand, a parallel finite element simulation can compute the reaction of the structure of the car to the air flow. The two parts are coupled using an appropriate communication layer.

