IN3200/IN4200: Chapter 5
Basics of parallelization

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

Objectives

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 in parallelization:

- The most common strategies for parallelization
- Simple theoretical insights into the factors that can hamper parallel performance

Why parallelize?

- We want to solve the 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.)

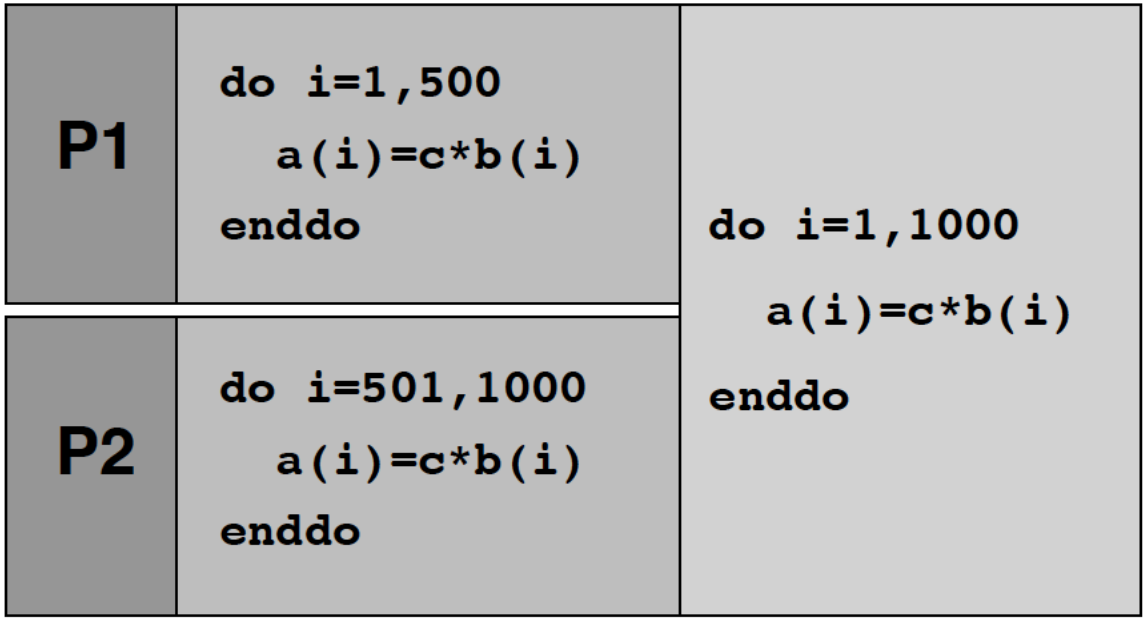


Figure 5.1: An example for medium-grained parallelism: The iterations of a loop are distributed to two processors P1 and P2 (in shared memory) for concurrent execution.

More examples of data parallelism

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):

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

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

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

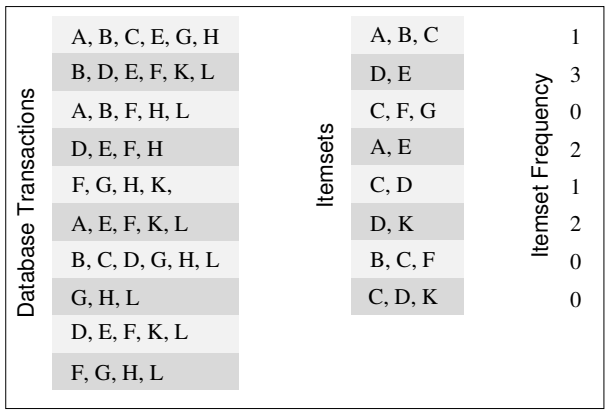
$$\text{Process 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

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

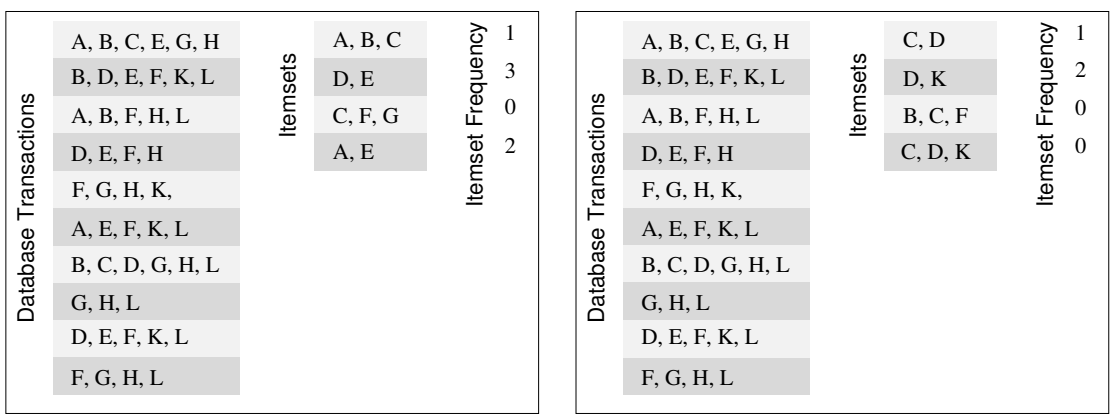
More examples of data parallelism

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

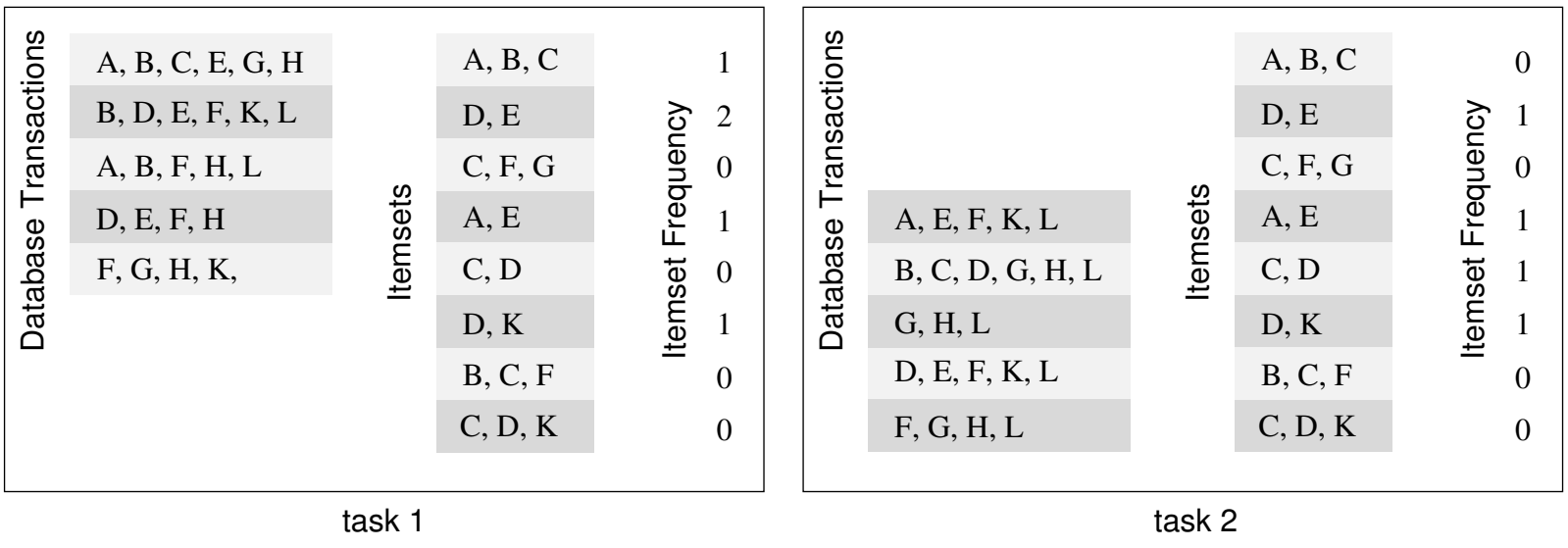
We have the following observations:

- 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

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

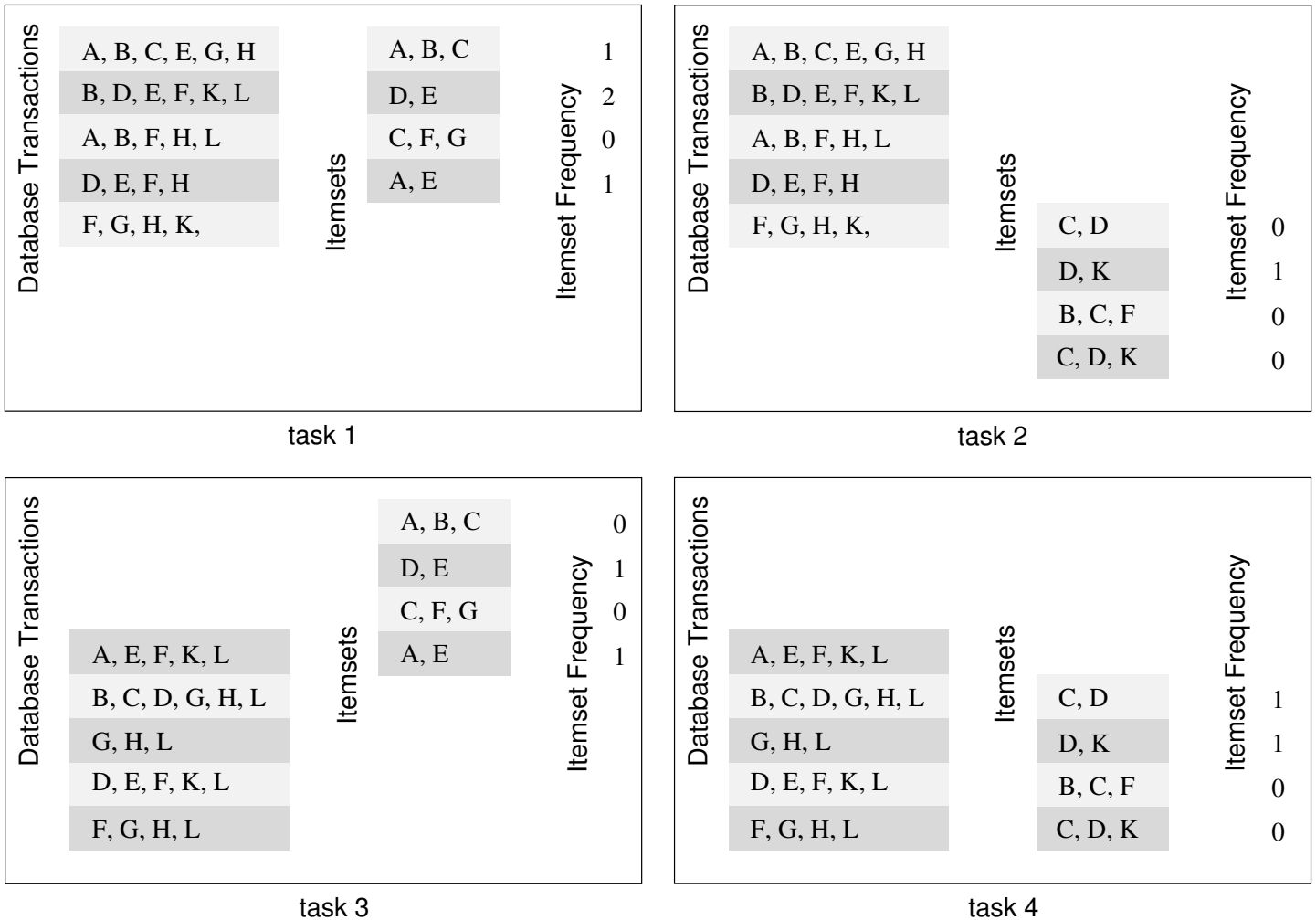


A. Grama, A. Gupta, G. Karypis, and V. Kumar: "Introduction to Parallel Computing", Addison Wesley, 2003

More examples of data parallelism

Input and Output Data Partitioning:

Partitioning both transactions and frequencies among the tasks

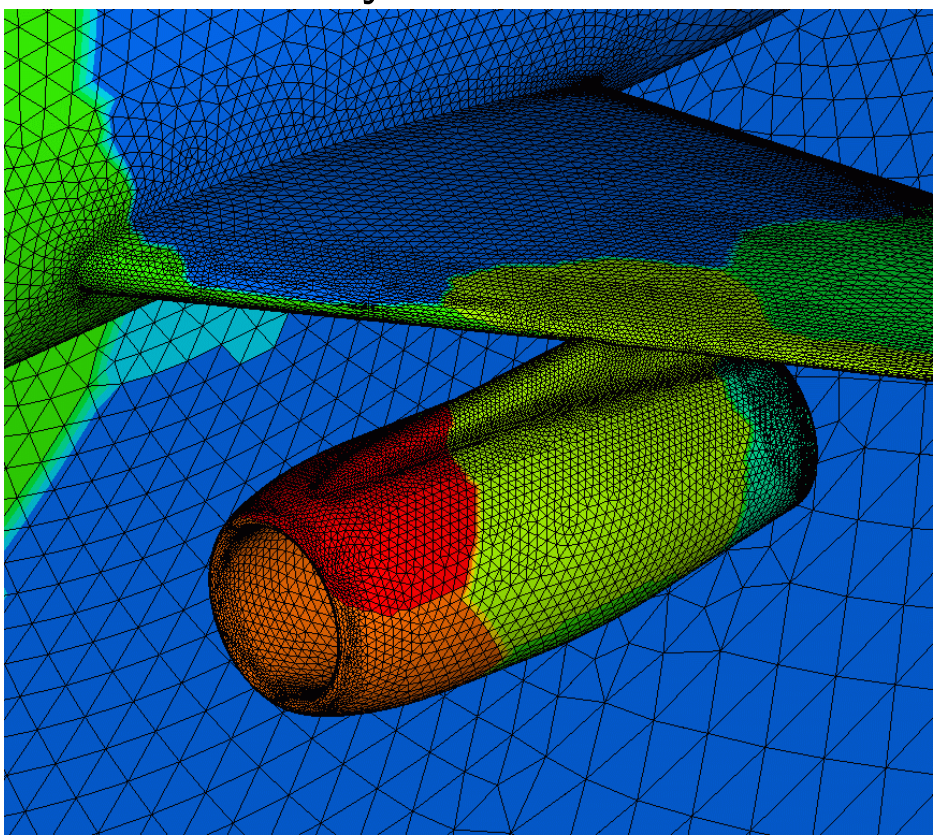


A. Grama, A. Gupta, G. Karypis, and V. Kumar: "Introduction to Parallel Computing", Addison Wesley, 2003

Example: Coarse-grained parallelism by domain decomposition

In case of a computational domain and 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.



https://fun3d.larc.nasa.gov/example-55.html

Domain decomposition & distributed memory

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

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

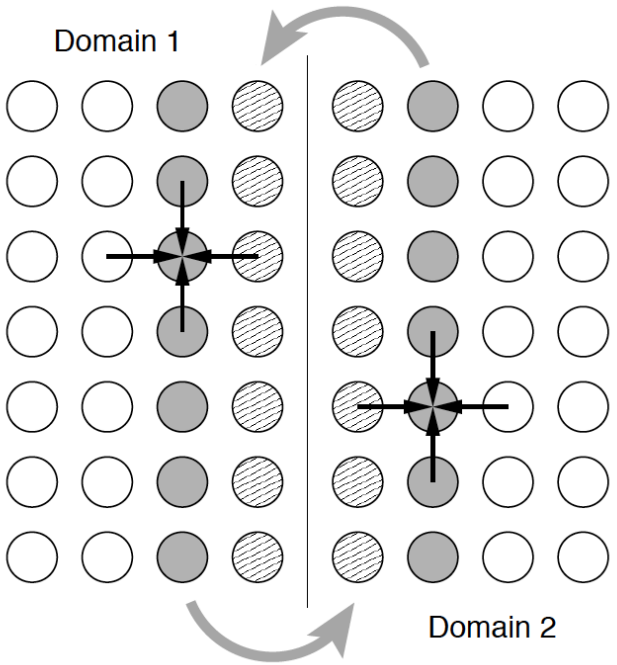


Figure 5.2: Using halo ("ghost") layers for communication across domain boundaries in a distributed-memory parallel Jacobi solver. After the local updates in each domain, the boundary layers (shaded) are copied to the halo of the neighboring domain (hatched).

More about domain decomposition

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

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

Impact of decomposition on communication overhead

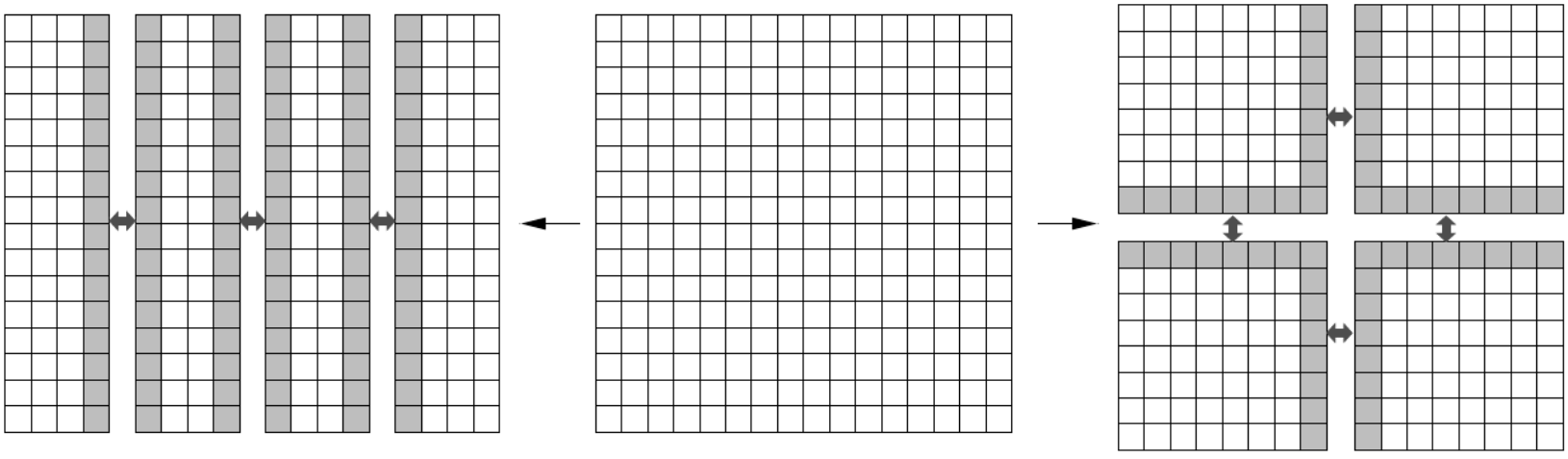
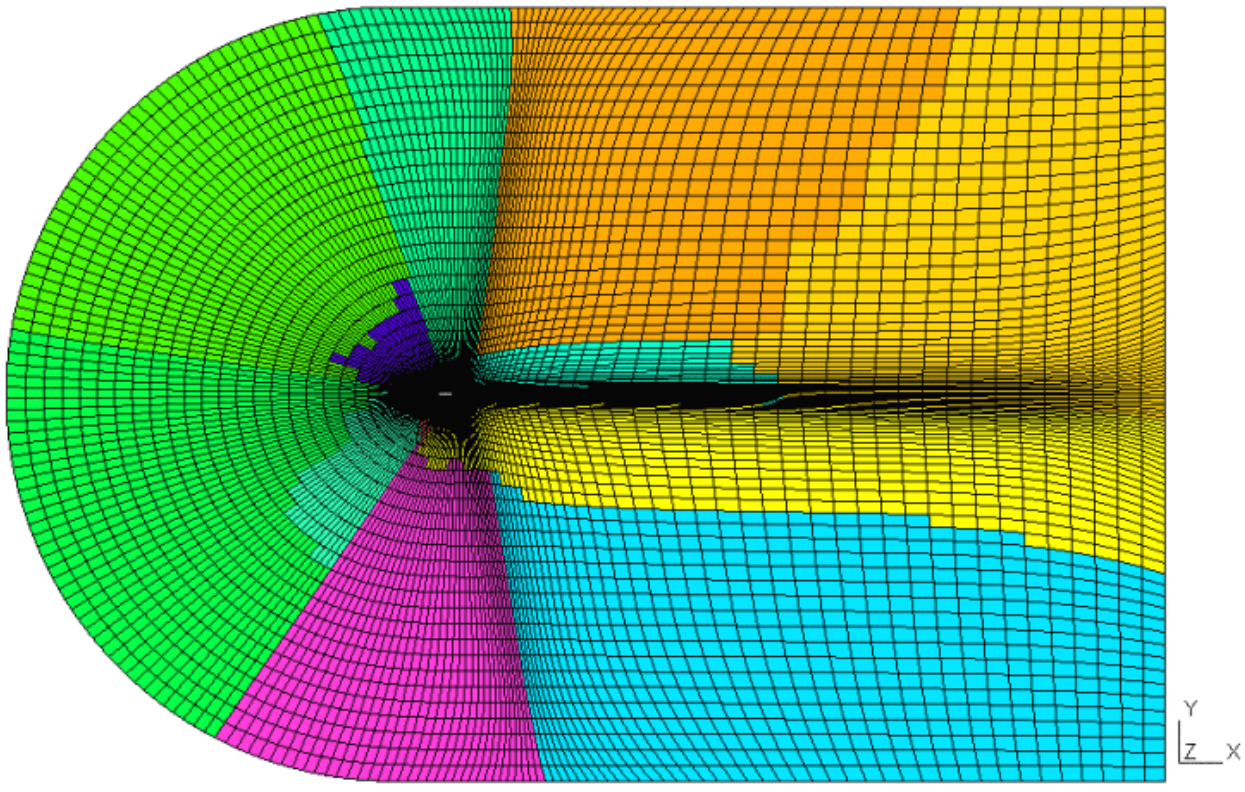


Figure 5.3: Domain decomposition of a two-dimensional Jacobi solver, which requires next-neighbor interactions. Cutting into stripes (left) is simple but incurs more communication than optimal decomposition (right). Shaded cells participate in network communication.

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").

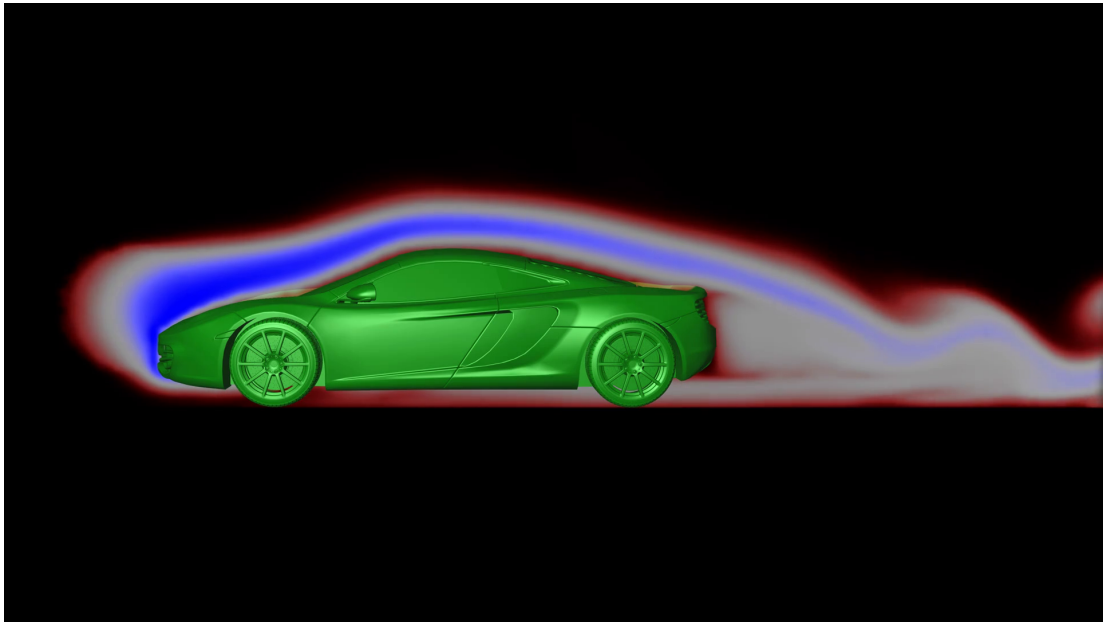
- From a pool of subtasks, the master dynamically assigns the workers with computational work.
- The master is also responsible for collecting results from the workers.

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 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.



Parallel scalability

The *ideal* goal: If a problem takes time T to be solved by one worker, we expect the solution time by using N identical workers to be T/N—a perfect **speedup** of N.

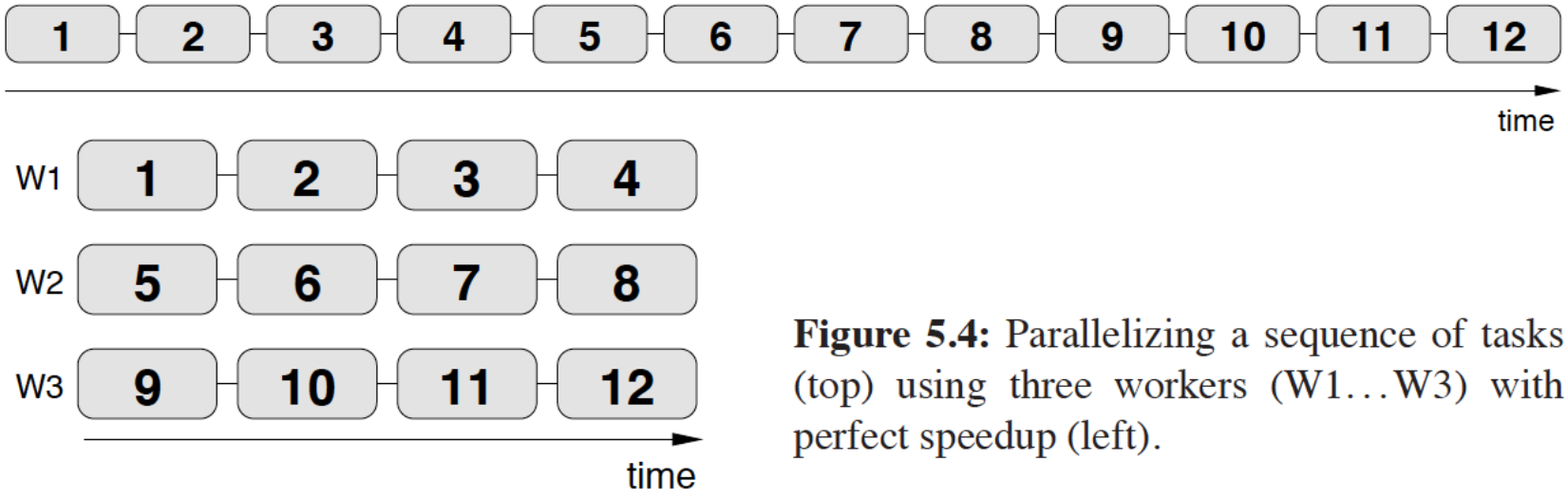


Figure 5.4: Parallelizing a sequence of tasks (top) using three workers (W1...W3) with perfect speedup (left).

Perfect speedup is often not achievable in reality, why?

Factors that limit parallel execution

Reasons for non-perfect speedup:

- Not all workers might execute their tasks equally fast, because the problem was not (or could not be) partitioned into equal pieces—load imbalance;
- There might be shared resources which can only used by one worker at a time—serialization;
- New tasks may arise due to parallelization, such as communication between workers—overhead.

Example of load imbalance

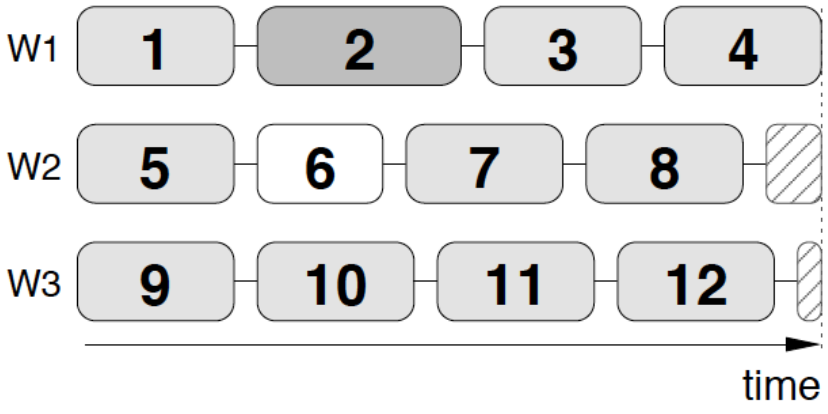
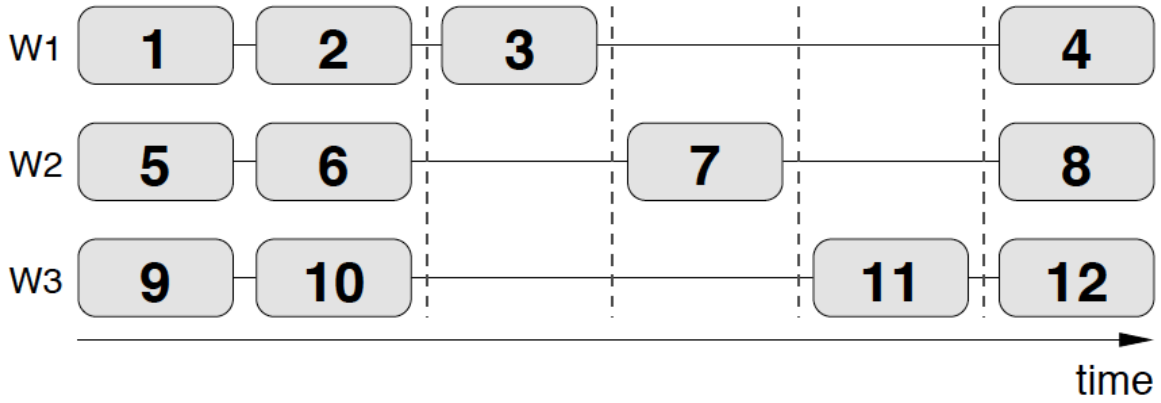


Figure 5.5: Some tasks executed by different workers at different speeds lead to *load imbalance*. Hatched regions indicate unused resources.

Example of serialization

Figure 5.6: Parallelization with a bottleneck. Tasks 3, 7 and 11 cannot overlap with anything else across the dashed "barriers."



Example of communication overhead

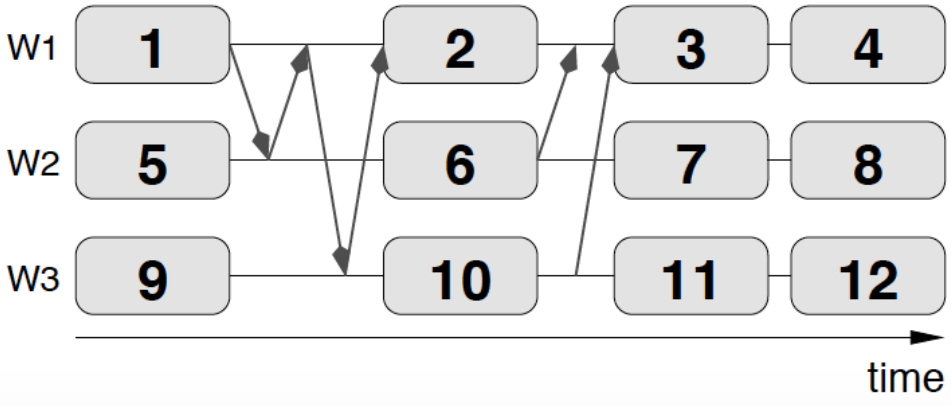


Figure 5.7: Communication processes (arrows represent messages) limit scalability if they cannot be overlapped with each other or with calculation.

Scalability metrics

How well can a computational problem be parallelized?

Scalability metrics help to answer the following questions:

- How much faster can a given problem be solved with N workers instead of one?
- How much more work can be done with N workers instead of one?
- What impact do the communication requirements have on performance and scalability?
- What fraction of the resources is actually used productively?

Strong and weak scaling

Starting point: The overall problem size ("amount of work") is *normalized* as

$$s + p = 1$$

where s is the serial, nonparallelizable fraction, p is the perfectly parallelizable fraction.

We can now define *strong scaling* and *weak scaling*, and study the relationship between single-worker serial runtime and multi-worker parallel runtime.

Strong scaling

Single-worker (serial) normalized runtime for a fixed-size problem:

$$T_{\mathrm{f}}^{\mathrm{s}} = s + p$$

Solving the same problem using N workers will require a runtime of

$$T_{\mathrm{f}}^{\mathrm{p}} = s + \frac{p}{N}$$

This is called **strong scaling**, because the total amount of work stays constant no matter how many workers are used.

Here, the goal of parallelization is minimization of time-to-solution for a given problem.

Weak scaling

For **weak scaling**, the goal is to solve an increasingly larger problem with more workers N.

More specifically, the total amount of work is scaled with some power of N

$$s + pN^{\alpha}$$

which means that single-worker runtime for the variable-sized problem **would have been** $T_{\mathrm{v}}^{\mathrm{s}} = s + pN^{\alpha}$.

Using N workers, the parallel runtime is

$$T_{\mathrm{v}}^{\mathrm{p}} = s + pN^{\alpha-1}$$

Here, we have also assumed that s doesn't grow with N.

The most typical choice is $\alpha = 1$, then $T_{\mathrm{v}}^{\mathrm{s}} = s + pN$ and $T_{\mathrm{v}}^{\mathrm{p}} = s + p$.

Simple scalability laws

How to calculate speedup?

$$\text{application speedup} = \frac{\text{serial runtime}}{\text{parallel runtime}}$$

or equivalently

$$\text{application speedup} = \frac{\text{parallel performance}}{\text{serial performance}}$$

where “performance” is defined as “work over time”.

Amdahl's law

For a fixed problem size $s + p = 1$, the application speedup ("scalability") is

$$S_{\mathrm{f}} = \frac{T_{\mathrm{f}}^{\mathrm{s}}}{T_{\mathrm{f}}^{\mathrm{p}}} = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{1-s}{N}}$$

This is "Amdahl's law"—maximum speedup is $1/s$ when $N \to \infty$.

Gustafson's law

The problem size is scaled with the number of workers N.

Recall that for $\alpha = 1$ we have $T_{\mathrm{v}}^{\mathrm{s}} = s + pN$ and $T_{\mathrm{v}}^{\mathrm{p}} = s + p$. Therefore the application speedup is

$$S_{\mathrm{v}} = \frac{T_{\mathrm{v}}^{\mathrm{s}}}{T_{\mathrm{v}}^{\mathrm{p}}} = \frac{s + pN}{s + p} = \frac{s + (1 - s)N}{1} = s + (1 - s)N$$

This is "Gustafson's law"—speedup can be arbitrarily large when $N \to \infty$.

Parallel efficiency

How effectively is the resource used by parallel program?

Parallel efficiency is defined as

$$\varepsilon = \frac{\text{speedup}}{N}$$

This will be a value between 0 and 100%.

Negative impact of load imbalance

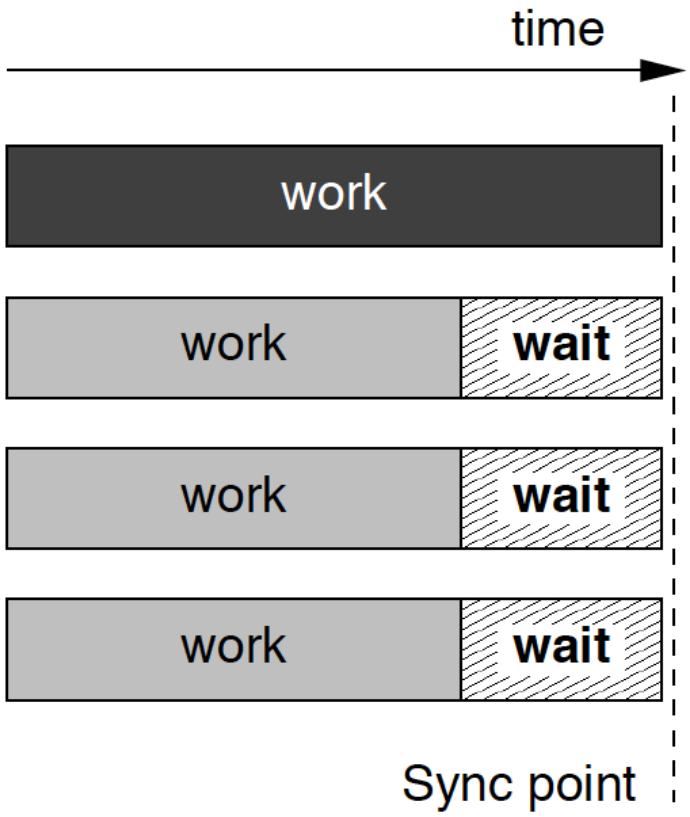


Figure 5.13: Load imbalance with few (one in this case) "laggers": A lot of resources are underutilized (hatched areas).

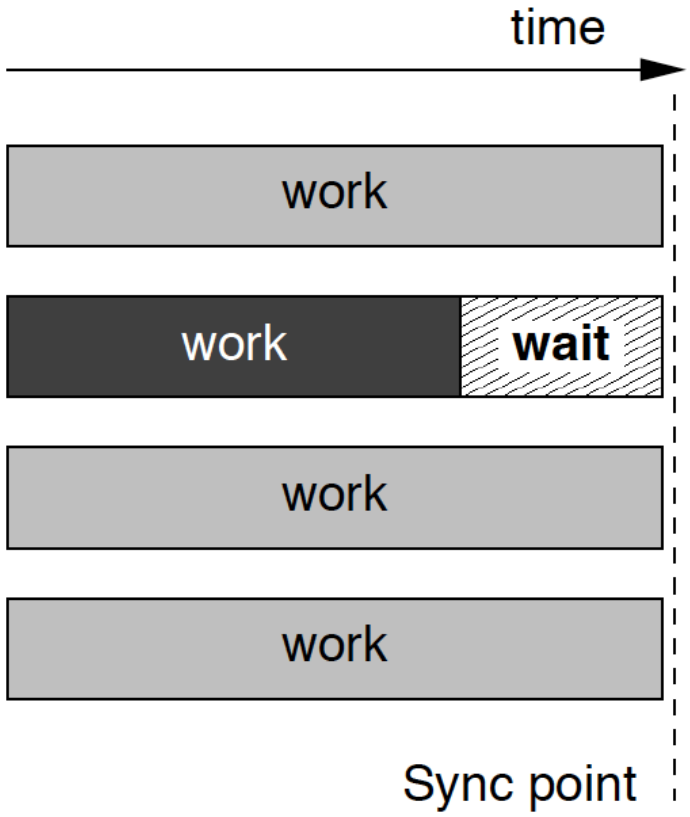


Figure 5.14: Load imbalance with few (one in this case) "speeders": Underutilization may be acceptable.