Advanced Methods for Scientific Computing (AMSC)
Lecture title: Introduction to Parallel Computing

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

Parallel Computing

In a parallel computing a problem is broken into several tasks that can be solved concurrently by different threads or processes. We have 2 types of parallel computers according to Flynn's taxonomy[1]:

SIMD Single instruction, multiple data. All processing units execute the same instruction on different data elements. (vector computers, GPUs);

MIMD Multiple instruction, multiple data. Each processing units operates on a different set of data and may perform different operations (multicore, computer clusters).

In this course we will be only concerned about MIMD architectures[2].

[1] In fact they are 4, but one is the scalar computer, and the other is not practical.

[2] We may have hybrid architectures, where we can mix the two paradigms in the same program

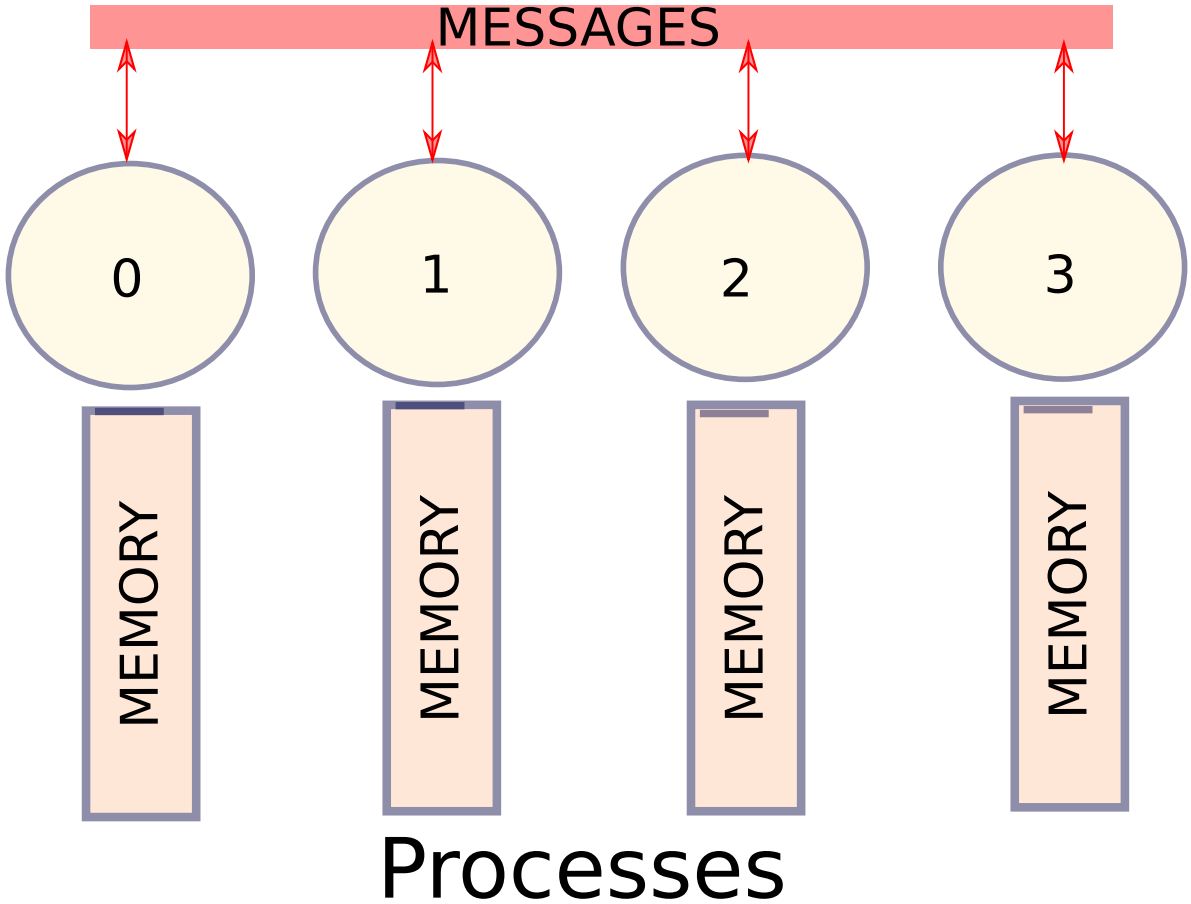
Shared or distributed memory

Another big distinction concerns how the computer memory is used by the different tasks.

- Distributed memory system. Every process[3] has its own memory, inaccessible to the other processes. Sharing of information among processes is made via messages.
- Shared memory system. Computer RAM memory is shared among tasks (normally assigned to different threads). Communication is made by accessing the common memory.
- Hybrid memory systems. Memory is distributed among processors, but each processor can run multiple cores that share the processor's memory.

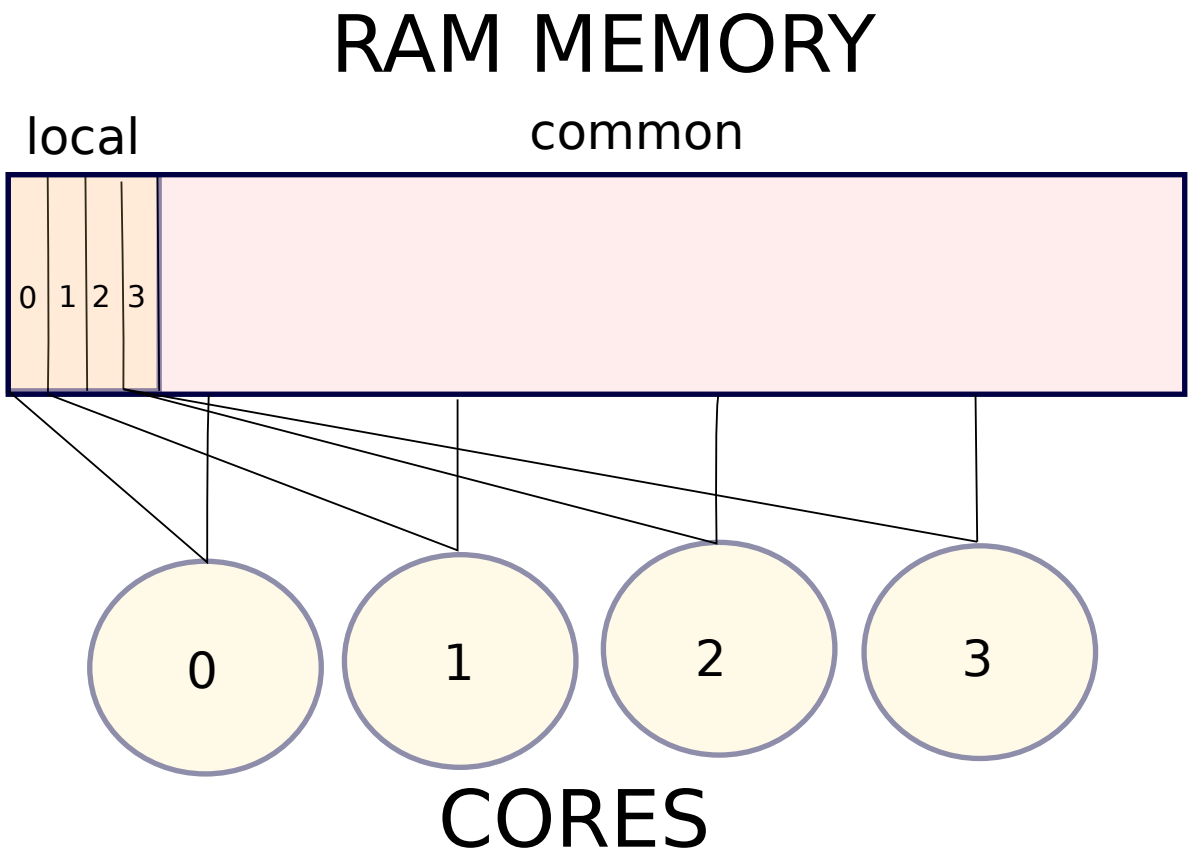
[3] normally in this case we talk of processes not threads since they may even reside on different machines!

Distributed memory system



Communication among processes is done explicitly by sending/receiving messages. The most important message passing protocol is MPI.

Shared memory system



Communication among cores (threads) is done implicitly by accessing the common memory resources. The most important protocol is OpenMP.

Parallel programming design

When porting a scalar program to a parallel environment, independently from the parallel protocol chosen, some steps are required:

1. Partitioning. Divide computation and data into small tasks focusing on those that can be executed in parallel.
2. Communication. Identify what communication needs to be carried out among the previously identified taks.
3. Aggregation. Combine tasks and communication into larger tasks. For instance, if task B should be executed after task A it makes sense to aggregate them in a single task.
4. Mapping. Assign the composite tasks identified in the previous point to processes/threads. Try to minimize communication (particularly in distributed memory environments).

Synchronization and deadlocks

An important aspects and cause of errors (and headaches) is the synchronization of tasks.

In a parallel code, there are moments where a task may have to wait until another task provides some data. Or, part of the code must be executed by just one task, while the others have to wait. Or, a part of the code should be executed by only one task at a time.

Failure of performing correct tasks synchronization may lead to a deadlock (all tasks idle) or to non-deterministic errors.

Data Decomposition

Often parallelism is realized by decomposing the data into chunks that are assigned to different tasks. Let's consider the simple situation of a for loop over the elements of a vector v of size n:

```
for (int i=0;i<n;++i)
{
   // parallelizable operations on v[i]
}
```

An idea is to partition v among the p available tasks. Two strategies may be followed

- Block decomposition. The first n_0 elements are assigned to task 0, the second n_1 elements to task 1 etc.
- Interleaved decomposition. Task 0 handles elements $(0, p, 2p, ..)$, task 1 elements $(1, p+1, 2p+1, ..)$ and so on.

Block Decomposition

We give more details on the block decomposition since is the most relevant in MPI setting. Given a vector of dimension n and p tasks we need to partition the elements among the tasks so that the work load is as evenly distributed as possible.

If the computational cost is uniform over the vector elements it means that each task should handle either $\lfloor n/p \rfloor$ of $\lceil n/p \rceil$, so that the difference is at most 1.

The questions we need to answer (in order of importance) are

- Which is the interval $[f_t, l_t[$ of the indexes handled by task t.
- Given the index i of the vector, which is the task t_i handling the corresponding element?

Types of block partitioning

We indicate with $r = n \bmod p$ the rest of the division between n and p (in C++ r=n % p). We recall that in C++ the division between integers n/p already gives $\lfloor n/p \rfloor$ directly!.

Grouped. The simplest to understand, the first r tasks get $\lfloor n/p \rfloor + 1$ elements, the others $\lfloor n/p \rfloor$[4]. , but the expression for t_i is complex:

$$f_t = t\lfloor n/p \rfloor + \min(t, r),\ l_t = (t+1)\lfloor n/p \rfloor + \min(t+1, r);$$

$$t_i = \min(\lfloor \tfrac{i}{(\lfloor n/p \rfloor + 1)} \rfloor, \lfloor \tfrac{i-r}{\lfloor n/p \rfloor} \rfloor).$$

Distributed: Better if the map $i \to t$ is needed:

$$f_t = \lfloor \tfrac{tn}{p} \rfloor,\ l_t = \lfloor \tfrac{(t+1)n}{p} \rfloor;$$

$$t_i = \lfloor \tfrac{p(i+1)-1}{n} \rfloor.$$

In Parallel/Utilities you have the file `partitioner.hpp` that defines two classes implementing the above formulae.

[4] An alternative is to have the first $p - r$ task take $\lfloor n/p \rfloor$ elements and the last r one more

Partitioning a matrix

Normally a matrix is stored in a contiguous buffer of memory following either a row-wise or a column-wise ordering (also called row major and column major, respectively).

For a $m \times n$ full matrix with row-wise ordering, the map from a generic element of indexed (i, j) to the index of the corresponding element in the data buffer is

$$(i, j) \rightarrow j + in.$$

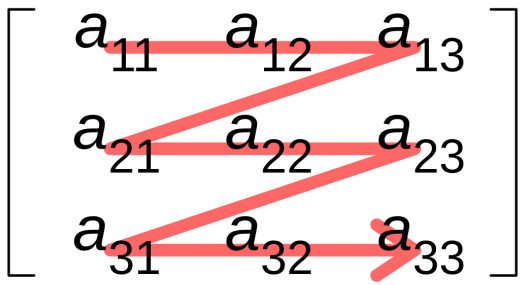
For a column-wise ordering the map is given by

$$(i, j) \rightarrow i + jm.$$

The most common partitioning scheme for a matrix is based on row or colum partitioning (for matrices with row-wise or column-wise ordering, respectively)

Row and column based ordering

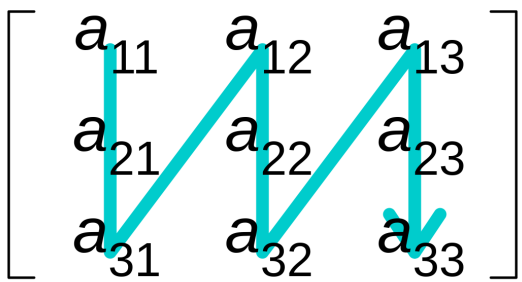
Row-major order



data buffer

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{21} & a_{22} & \cdots \end{bmatrix}$$

Column-major order



data buffer

$$\begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{12} & a_{22} & \cdots \end{bmatrix}$$

Partitioning a matrix

With row/column block partition we distribute to each task t $\lfloor m/p \rfloor (+1)$ rows (rep. $\lfloor p/p \rfloor (+1)$ columns), where the $+1$ accounts for the fact that the integer division may have a rest.

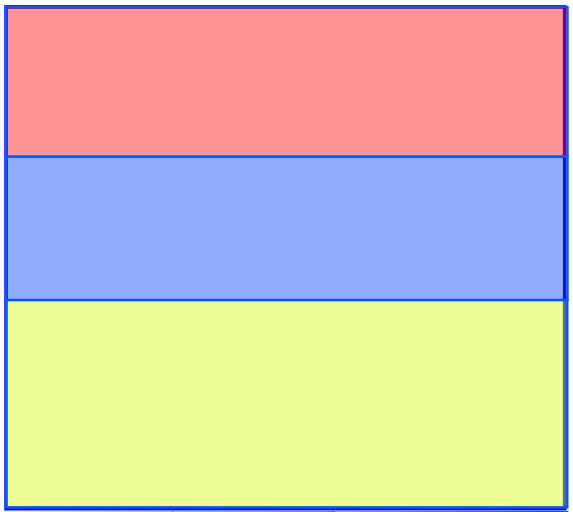
If $l(l, t)$ is the starting index for a block decomposition of a vector of lenght l on task t task, the starting index k of element (i, j) in the data buffer of the matrix is:

- row-wise ordering and partitioning: $k = n \cdot l(m, t)$
- column-wise ordering and partitioning: $k = m \cdot l(n, t)$

Therefore, having the tool to block partition a vector we have also the tools to block partition a matrix if the block partition is consistent with the ordering.

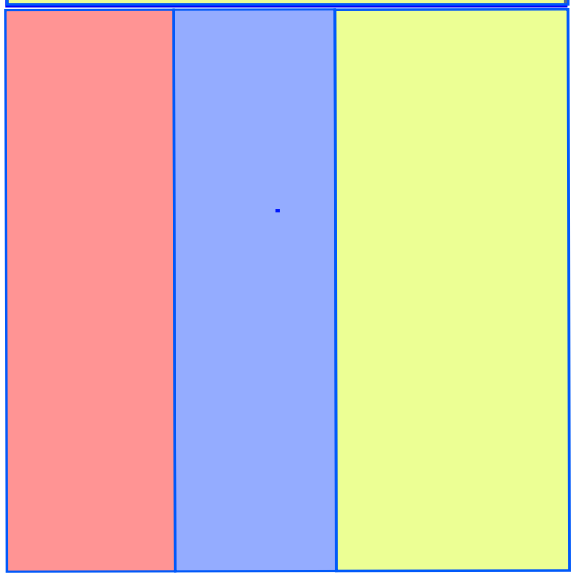
There are other matrix partitioning strategies equilibrate the elements among tasks better when the number of row/columns is not a multiple of p, but are less used in practice.

Block partition of a matrix



row block partitioning

data buffer



column block partitioning

data buffer

Normally, if you use column block partitioning, for efficient matrix*vector you should partition also vectors accordingly

Speed-up and Efficiency

Speed up S is simply defined as the ratio between the computing time on a single process/thread t_s and that in the parallel implementation, t_p:

$$\boxed{S = \frac{t_s}{t_p}}$$

If we have p parallel tasks the optimal speed-up is p, but that value is never reached in practice. Many factors affect the efficiency, defined as

$$\boxed{E = \frac{S}{p} = \frac{t_s}{pt_p}},$$

parts of the program that cannot be made parallel, unbalanced workload among tasks, the cost of communication...

Factors affecting speedup and Amdahl's law

Let n and p be the size of the problrma at hand and the number of parallel tasks. We indicate with

- $\sigma = \sigma(n)$ The time spent in the inherently sequential portion of the program;
- $\phi = \phi(n)$ The time spent in the parallelizable portion of the program;
- $\kappa = \kappa(n, p) = l(p) + c(n, p)$ The time spent for communication among tasks, composed of the latency l and the data communication c.

We have $t_s = \sigma + \phi$ and $t_p = \sigma + \phi/p + k$, then

$$\boxed{S = \frac{\sigma + \phi}{\sigma + \phi/p + \kappa} \leq \frac{1}{r + (1 - r)/p} \leq \frac{1}{r}}$$

with $r = \sigma/(\sigma + \phi)$ being the non parallelizable fraction of the code: Amdhal's law.

Weak and strong scalability

Scalability is related to how parallel efficiency behaves varying the size of the problem and the number of parallel processes.

- A program is strongly scalable if, for a fixed dimension of the problem, the efficiency does not decrease when increasing the number of cores.
- A program is weakly scalable if the efficiency does not decrease when the problem dimension is increased proportionally to the number of cores.

Strong scalability is normally obtained only for embarrassingly parallel programs. It is normally more reasonable to aim at weak scalability.

Weak scalability

We have

$$E = \frac{1}{pr + (1 - r) + p\frac{\kappa}{\sigma+\phi}}$$

To have weak scalability we want $E \simeq$ const when $p = O(n)$. This is reached, for instance, if $r = O(n^{-1})$ and $\frac{\kappa}{\sigma+\phi} = O(n^{-1})$. We have this situation if $\sigma = O(n^s)$ while $\phi = O(n^{s+1})$ and $k = O(n^s)$.

For example, if we take $s = 1$, a sufficient condition for weak scalability is having the parallel portion of the code growing quadratically with respect to the size of the problem n, while the scalar part and the communication cost growing at most linearly.

Other measures of parallel performance

Besides speedup and efficiency there are other measures of parallel effectiveness.

For the sake of time we omit a full description here. You may find a nice discussion in Chapter 7 of Parallel Programming in C with MPI and OpenMP of M.J. Quinn, McGrow Hill, 2003.

We mention here only the Karp-Flatt metric: for $p > 1$

$$K_f = \frac{1/S - 1/p}{1 - 1/p} = \frac{p - S}{Sp - S}$$

is the experimental serial fraction.

Use of the Karp-Flatt metric

For a given parallel code, let $K_f(p)$ be the experimental serial fraction when the code is run with p parallel tasks and $E(p)$ the corresponding efficiency. Unless the code is embarrassingly parallel, we expect a decrease of E with the increase of p.

If $K_f(p)$ remains approximately constant (or even increases), it means that a loss of parallel efficiency is mainly due to the communication cost κ. If instead, $K_f(p)$ increases as p increases, then the loss of efficiency is due to the parallel algorithm that loses effectiveness for large number of processor.

Timing parallel code

Timing portions of a code can be performed either by the standard C++ functions provided in the header <chrono>, or specialized tools. For instance, MPI provides specific tools for timing programs. In both cases normally timing is performed by one process/thread.

Timings should be taken when the computer is not running other heavy tasks, and the code should be run several times using the same parameters.
Eventually, the smallest time should be taken: system interrupts may only increase the time used by your program, so the minimal value is what is nearer to the effective computational cost of your code.

Serialization

When dealing with message passing, a critical issues, particularly for complex user-defined types, is serialization.

Serialization is the process of packing the content of an object in a contiguous buffer of memory, so that it can be transferred and reconstructed. Serialization is also critical when storing data. If the data is already stored contiguously (like in a std::vector or std::array) it is easy. In general, it is not.

The Boost serialization library provides tools to build serializable objects. Here we do not go into details, yet we discuss trivially copyable classes, since in this case serialization is trivial.

Trivially copyable classes

Aggregates and in general trivially copyable objects can be easily serialised.

Luckily, the set of class type that can be trivially serialised is larger than just aggregates, they are the trivially copyable classes and we have a tool to test it!.

The tool is std:: is_trivially_copyable_v <T>, included in the <type_traits> header file, which is true if T is trivially copyable. Let's recall what it means, with an example that uses memcpy, a tool that copies memory buffers.

```
void * memcpy (void * dest, const void * source, size_t n);
```

Trivially copyable classes

```
#include <type_traits>
#include <cstring> // for memcpy
....
MyClass a;
//... fill a with values
MyClass b;
if constexpr (std::is_trivially_copiable_v<MyClass>)
// I copy a into be as a memory buffer
std::memcpy(&b,&a,sizeof(MyClass));
else
// I have to do something different
// maybe just b=a
```

This feature may be handy, since MPI communications use buffers to transmit data between processes.

Side-effects

An important concept is that of side-effect. A callable object has a side-effect if it modifies an object outside the scope of the function in an indirect way. i.e. not through the returned values.
In this case, you need to be careful in a parallel implementation!

Let's consider the following Cont7 functor that counts how many times it has been called with a 7:

```
struct Count7{
    void operator ()(int i){if (i==7) ++count7;}
    int count7=0;
};
...
Count7 c7;
for (auto i:x) c7(i);
std::cout<<" The number of 7 in x are "<<c7.count7;
```

Each call may modify the internal state of the Count7 object. So it has a side-effect.

Side-effects and parallel programming

In a distributed memory paradigm like MPI, each process has its own copy of a Count7 object, which counts the 7s in the portion of the vector assigned to it. To have the total number you have to sum them up!

If you use a shared memory paradigm instead, like threads or openMP, you must ensure that different threads do not access the counter at the same time:

```
std::vector<int> x;
// fill x with values
Count7 c7; // I want to count the 7
#pragma omp parallel for //start parallel loop
for (auto i:x) c7(i);
std::cout<<" The number of 7 is "<<c7.count7;
```

In a multithread environment, the answer is almost surely wrong and different at each run of the code. You have to make the counter update atomic!