

# Background Title

# Foreground



SCIENTIFIC &  
DATA-INTENSIVE COMPUTING

Luca Tornatore - I.N.A.F. 

**Advanced HPC 2024-2025 @ Università di Trieste**

# Assignment

## The Stencil method




SCIENTIFIC &  
DATA-INTENSIVE COMPUTING



DATA SCIENCE &  
ARTIFICIAL INTELLIGENCE

Luca Tornatore - I.N.A.F. 

**Basic HPC 2024-2025 @ Università di Trieste**



# Some elements on BlaBlaBla



**Università di Trieste**  
**HPC 2024-2025**

Luca Tornatore  
I.N.A.F.





# Introduction

v1.2 06.26

You are requested to parallelize a *stencil computation* starting from a serial version (or from a parallel template where many components are missing).

The example application is the classical *heat equation*:

$$\frac{\partial u(t, \vec{x})}{\partial t} = \alpha \nabla^2 u(t, \vec{x}) \quad [1]$$

( $\alpha$  is the heat diffusivity and  $u$  the energy at position  $\vec{x}$  and time  $t$ ) which in two dimensions reads as

$$\frac{\partial u(t, x, y)}{\partial t} = \alpha \left( \frac{\partial^2 u(t, x, y)}{\partial x^2} + \frac{\partial^2 u(t, x, y)}{\partial y^2} \right). \quad [2]$$

Discretizing on  $m \times 1$  grid the values of the energy  $U_{m,\ell}^n$  and adopting an explicit finite-difference method, yields

$$U_{m,\ell}^{n+1} = U_{m,\ell}^n + \frac{\alpha \Delta t}{\Delta x^2} (U_{m-1,\ell}^n + U_{m+1,\ell}^n - 2U_{m,\ell}^n) + \frac{\alpha \Delta t}{\Delta y^2} (U_{m,\ell-1}^n + U_{m,\ell+1}^n - 2U_{m,\ell}^n) \quad [3]$$

in which the values at time  $t_{k+1}$  at position  $(i,j)$  depends on previous values at time  $t_k$  at the points  $(i,j)$  itself and the four surrounding points  $(i-1,j)$ ,  $(i+1,j)$ ,  $(i,j-1)$  and  $(i,j+1)$ .

The approximation expressed by [3] is known as the five-points stencil.

There are several different ways to numerically solve equation [1], but here we are interested in this peculiar case since many methods ends in being stencil-based.

Higher-order approximations would result in a larger stencil (9-points, 25-points, and so on)

# Introduction

In the folder **Assignment/** you find:

```
Assignment/  
├── include  
│   ├── stencil_template_parallel.h  
│   └── stencil_template_serial.h  
├── slides.pdf  
└── src  
    ├── stencil_template_parallel.c  
    └── stencil_template_serial.c
```

the template for a parallel code

the template for a serial code

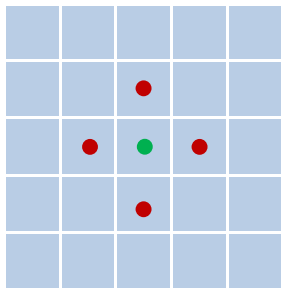
the template for the parallel code already contains the initialization, some memory allocation, and the skeleton of the main loop.

However, it already proposes data structures.

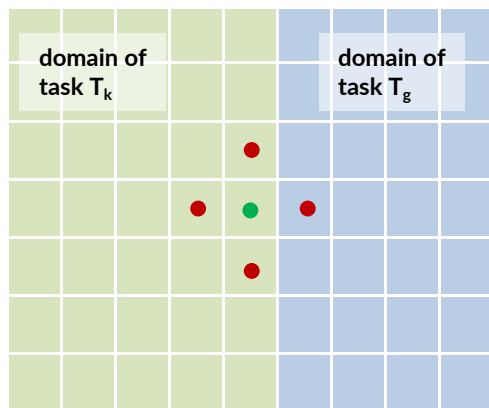
You can start from fresh from the serial version, which is a complete very simple version.

**NEW:** I added a template batch script for job submission, it is named **go**.

# Introduction

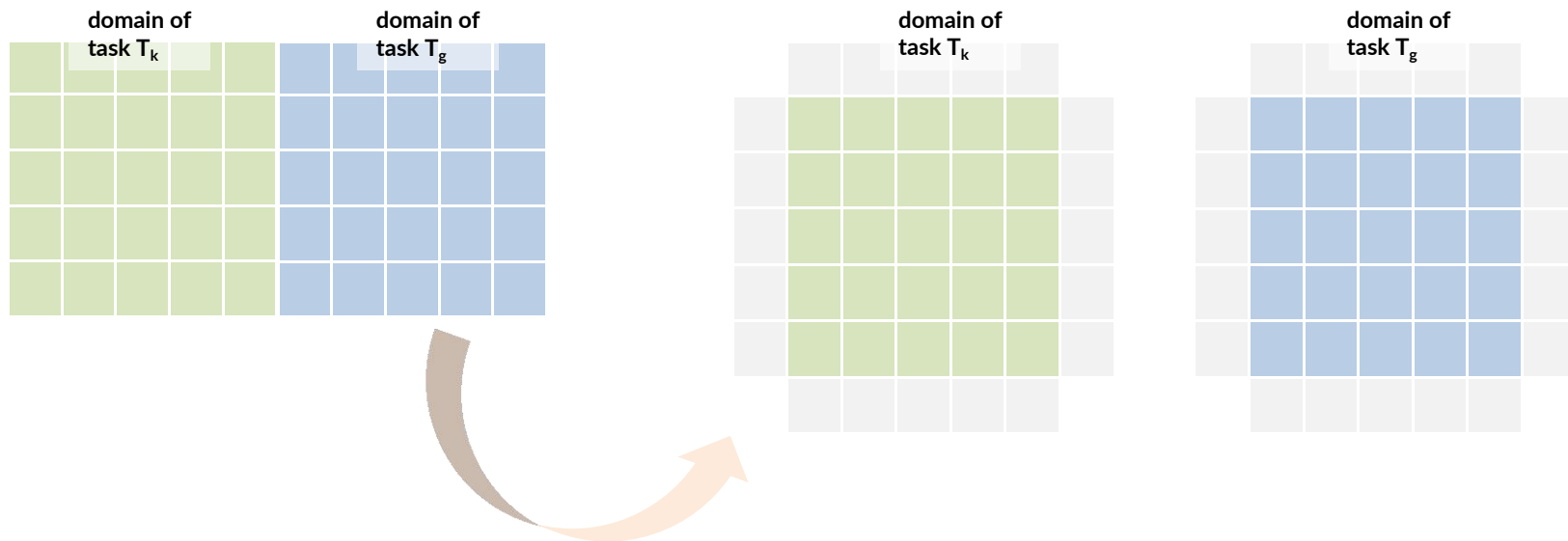


The new value of the green point depends on its old values and on the values of four surrounding points, i.e. it is “local”. However, if you are parallelizing the algorithm, the points at the border of the computational domain of task  $T_k$  belong to another task  $T_g$ , unless they are at the border and the boundary conditions are not periodic.



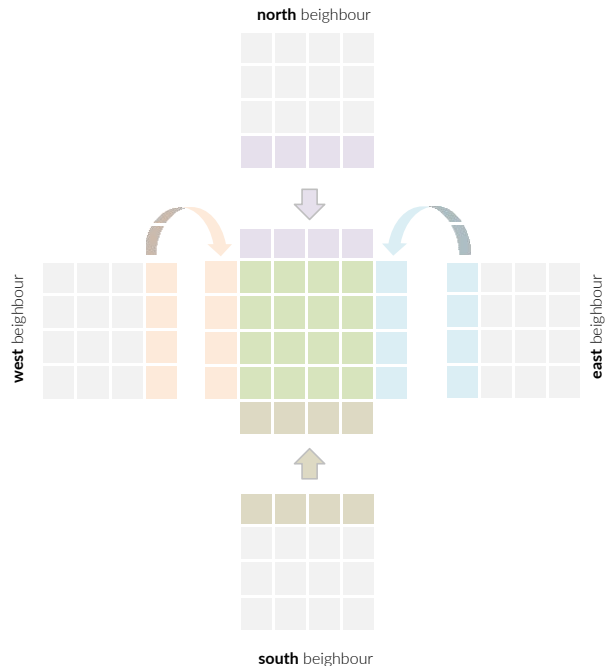
Both tasks  $T_k$  and  $T_g$  need to know the values of the points at the border of their neighbours (for a 5-points stencil only a layer of 1 point is sufficient)

# Introduction



Every task then needs an additional layer that contains the data from its north, east, south and west neighbours.

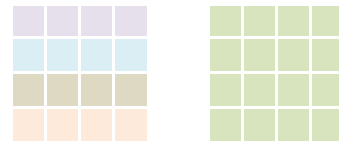
# Introduction



At every timestep, the updated values must be communicated to/from the involved tasks.

Every task may either embed its local grid patch into a larger one, or have separated buffers.

**Note:** we have not seen MPI derived data types during the lectures. Hence, you'll be forced to move data explicitly from/to the east and west buffers.

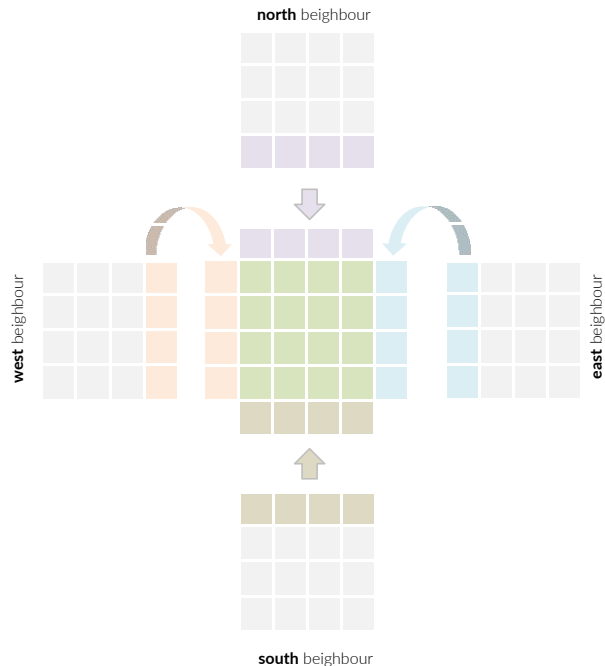


Having separated buffers makes the communication easier, but you still have to move data to/from the buffers (which in any case is still true for east and west, if you do not use derived data types)

When the local grid is embedded into a larger one, checking for the boundaries in the update loop is easier. But - if you do not build a derived vector data type - you need to specifically treat non-contiguous buffers (in C, the east and west buffers) when communicating.



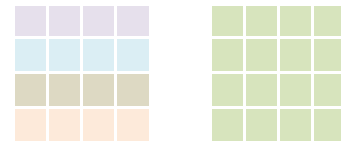
# Introduction



At every timestep, the updated values must be communicated to/from the involved tasks.

Every task may either embed its local grid patch into a larger one, or have separated buffers.

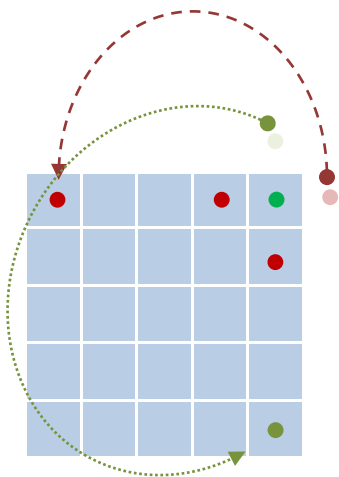
**Note:** we have not seen MPI derived data types during the lectures. Hence, you'll be forced to move data explicitly from/to the east and west buffers.



Having separated buffers makes the communication easier, but you still have to move data to/from the buffers (which in any case is still true for east and west, if you do not use derived data types)

When the local grid is embedded into a larger one, checking for the boundaries in the update loop is easier. But - if you do not build a derived vector data type - you need to specifically treat non-contiguous buffers (in C, the east and west buffers) when communicating.

# Introduction



When the boundary conditions are *periodic*, your plate behaves like it is infinite and with no borders.

# Introduction

All in all, the parallel code will perform the following steps:

## 1. initialization

- read arguments from the command line:  $(x,y)$ -size of the plate, the number of iterations  $N_{iter}$ , the number of heat sources  $N_{heat}$ , whether the boundary conditions are periodic.
- determine the MPI tasks grid: how you decompose the computational domain into a grid of  $N_x \times N_y$  tasks
- determine the neighbours of every MPI tasks (depends on the decomposition and on the boundary periodicity)

## 2. update loop

- for  $N_{iter}$ 
  - inject energy from the sources.
  - update the local grid.
  - exchange borders with the neighbours, if any.

# Introduction

All in all, the parallel code will perform the following steps:

## 1. initialization

- read arguments from the command line: (x,y)-size of the plate, the number of iterations  $N_{\text{iter}}$ , the number of heat sources  $N_{\text{heat}}$ , whether the boundary conditions are periodic.
- determine the MPI tasks grid: how you decompose the computational domain into a grid of  $N_x \times N_y$  tasks
- determine the neighbours of every MPI tasks (depends on the decomposition and on the boundary periodicity)

## 2. update loop

for  $N_{\text{iter}}$

- inject energy from the sources.
- update the local grid.
- exchange borders with the neighbours, if any.

hint: is there any possible, at least partial, overlap between these two dsteps?

# Introduction

All in all, the parallel code will perform the following steps:

## 1. initialization

- read arguments from the command line:  $(x,y)$ -size of the plate, the number of iterations  $N_{iter}$ , the number of heat sources  $N_{heat}$ , whether the boundary conditions are periodic.
- determine the MPI tasks grid: how you decompose the computational domain into a grid of  $N_x \times N_y$  tasks
- determine the neighbours of every MPI tasks (depends on the decomposition and on the boundary periodicity)

## 2. update loop

- for  $N_{iter}$ 
  - inject energy from the sources
  - **update the local grid**
  - exchange borders with the neighbours, if any

**These ingredients are already present in the templates**

# Requirements

- (A) build a parallel implementation of the 5-points stencil, based on the provided templates
- (B) parallelization must be with MPI *and* OpenMP (for the update loop of the local grid)
- (C) instrument your code so to know how much time is spent for computation and for communication
- (D) perform a scalability study (see next slides)



# Requirements

(D) perform a scalability study, **new: details**

- **openmp** ■ use 1 MPI task and scale the nr. of threads from 1 up to filling the entire code (there are 56 cores/socket, 2 sockets; it is not necessary to have 112 different runs, a series like 1,2,4,8,16,32,56,84,112 is ok).
- **MPI** ■ from the openmp scaling find the best threads-per-task ratios and elect your preferred choice; for instance, let's say that you choose 8 MPI tasks and 14 openmp threads per node.  
That given, perform strong and weak scaling using nodes as reference, ranging from 1 up to 16 or 32 nodes (1,2,4,8,16,32).

Here you find the configuration of the Leonardo partitions:

<https://leonardo-supercomputer.cineca.eu/hpc-system/#jump-partition>

# Requirements

## new: advices for submitting the jobs and tune the run time

reminder: you find a batch file template as [Assignment/go](#)

The budget that you've got from UniTs is limited and has to be shared among all of you. Currently it amounts to **50k** core-hours.

Let me explain the meaning of it by examples, referring to a node of LEONARDO DCGP (2 sockets, 56 cores each)

- 1 thread using 1 core for 1h = 1 core-hour
- 112 threads on a node, one thread per core, for 1h = 112 core-hours, or 1 node-hour
- 14 MPI tasks, 8 threads each on a node, for 1h = 112 core-hours, on 1 node-hour

Let's derive some estimate from this.

In the following, I'm assuming that for the MPI scaling

- you scale by nodes, using 1,2,4,8,16 nodes (go beyond if you feel like)
- $f$  is the efficiency of your scaling

# Requirements

**new: advices for submitting the jobs and tune the run time**

**reminder: you find a batch file template as Assignment/go**

In the following, I'm assuming that for the MPI scaling

- you scale by nodes, using 1,2,4,8,16 nodes (go beyond if you feel like)
- $f$  is the efficiency of your scaling

The amount of time needed for a strong scaling is, roughly:  $T_{\text{total}}^{\text{strong}} = T_0 \times \sum_{i=0}^k \frac{T_{n_i}}{T_0} \times n_i$

where  $k$  is the number of points you use (in our case, 5:  $n_1=1$ ,  $n_2=2$ ,  $n_3=4$ , ...)

$T_0$  is the serial time,  $T_{n_i}$  is the run-time with  $n_i$  nodes.

If we assume that your code scales as  $T_{n_i} \simeq \frac{T_0}{f n_i} \Rightarrow \frac{T_{n_i}}{T_0} \simeq \frac{1}{f n_i}$

then we have  $T_{\text{total}}^{\text{strong}} \simeq T_0 \times \frac{k}{f}$  node-hours, which translates in  $112 \times T_{\text{total}}^{\text{strong}}$  core-hours.

Similarly, for the weak scaling,  $112 \times T_{\text{total}}^{\text{weak}} \simeq 112 \times T_0 \times f k$  core-hours

# Requirements

**new: advices for submitting the jobs and tune the run time**

**reminder: you find a batch file template as Assignment/go**

Translating that in real numbers, assuming a poor  $f \sim 0.3$  for strong scaling and  $f \sim 1.3$  for weak scaling

time needed for **strong scaling** if your serial time<sup>(\*)</sup> is 3minutes ~ 90 core-hours

time needed for **weak scaling** if your serial time<sup>(\*)</sup> is 3minutes ~ 36 core-hours

for a total of **~120 core-hours**.

Rounding to **150 core-hours per each of you** to include some tests (which I suggest you do limiting the amount of cores and memory, and commenting the `#SBATCH --exclusive` line in the batch file), the budget should be sufficient for ~330 students.

Since you are ~50, that leaves enough room. I leave to each of you to judge.

Using the command

**saldo -b --dcgp** you can inspect how much time has been consumed

<sup>(\*)</sup> besides the performance of your code, you control the runtime by the problem size and the number of iterations

# Requirements

**new: advices for submitting the jobs and tune the run time**

**reminder: you find a batch file template as Assignment/go**

-- NOTE --

At the moment of writing, 26th June, the budgte has been moved to LEONARDO DCGP.

As such, you can see it using

**saldo -b --dcgp**

and it amounts to 50kcore-hours.

Then, the template script for submission is

**go\_dcgp**

# | Scalability

I remind you that “scalability” is the capability of your code to efficiently use different amount of computational resources(\*) when

1) the problem’s size is constant, which is called **strong scaling**

2) the problem’s size per resource is constant, which is called **weak scaling**

(\*) “Resources” may be MPI tasks and/or threads (then basically how many cores you use) or nodes, if you always use entire nodes (which is oftne the case)



# | Strong Scaling

Considering a given problem, we have seen in the lectures that we expect the time-to-solution to scale as  $\propto 1/n$  if  $n$  is the amount of computational resources, for a perfect scaling.

The difference between the real scaling and the perfect scaling is due to either the intrinsic serial fraction  $f$  of the problem, or to parallel overhead  $k$ .

If your problem has a serial fraction, then your speed-up has an hard limit:

$$S_p = t(1, s) / t(n, s) \rightarrow 1/(1-f)$$

where  $t(n, s)$  is the run-time for a size  $s$ , using  $n$  resources (which I remind you is called “the Amdahls’s law”).

# | Strong Scaling

Then, when you test a code for its strong scaling capability, as first analyze your algorithm and individuate where you lose parallelism by its very nature (and, at the discussion, try to imagine a different algorithm or a way to express more parallelism).

Then, check whether your scaling results match with your analysis, and how the run-time scales once you purge the inherently serial part.

That is why requirement c) is about instrumenting your code so that you can check the timing of different sections *and* of the communications.

At the end, you should be able to characterize your code, and discuss whether any loss of parallelism is expected because it is due to the algorithm, or it is due to your implementation.

# | Strong Scaling

Given that you expect a behaviour, the best way to present your results is *not* by plotting absolute times.

In fact, human eye is not good at judging non-linear behaviours, and in case you need to plot real timings, always superimpose the ideal scaling to your data.

The best way to report about scalability is to plot the **Speedup**, as defined before and in the lectures and that it is expected to be linear, or the efficiency, which is defined as

$$\text{Eff}(n, s) = \text{Sp}(n, s) / n$$

which is expected to be  $\sim 1$  for an ideal behaviour.

# | Strong Scaling

Given that you expect a behaviour, the best way to present your results is *not* by plotting absolute times.

In fact, human eye is not good at judging non-linear behaviours, and in case you need to plot real timings, always superimpose the ideal scaling to your data.

The best way to report about scalability is to plot the **Speedup**, as defined before and in the lectures and that it is expected to be linear, or the efficiency, which is defined as

$$\text{Eff}(n, s) = \text{Sp}(n, s) / n$$

which is expected to be  $\sim 1$  for ideal behaviour.

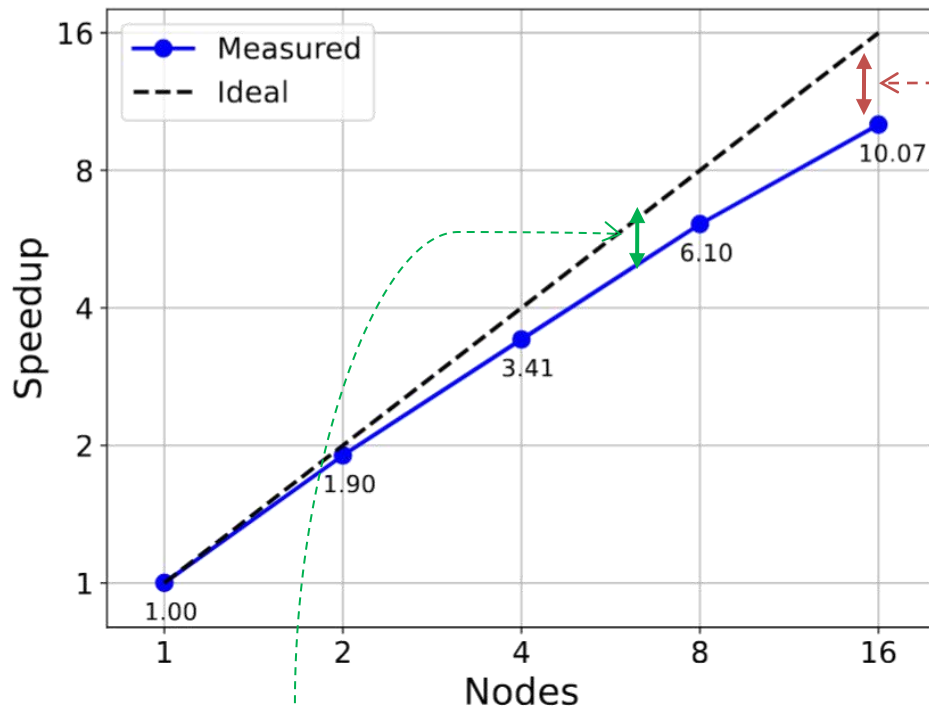
# Strong Scaling

At the end, you end up with a plot that is similar to the one on the right.

on the x-axis there are the computational resources (in this case the nodes, not the MPI tasks or the threads), and on the y-axis there is speed up.

Note that the ideal scaling is reported to guide the eye.

There will always be some lost of efficiency for a large scaling factor, because for a large-enough amount of resources the parallel overhead will be dominant. There is not a formal definition of this threshold, since that is very much dependent on the problem and the algorithm. However I would say for good code that should happen beyond  $16\times$  ...



gap due to inherent non-parallelism, communication, overhead due to inefficiency or to parallelization itself?

# Strong Scaling

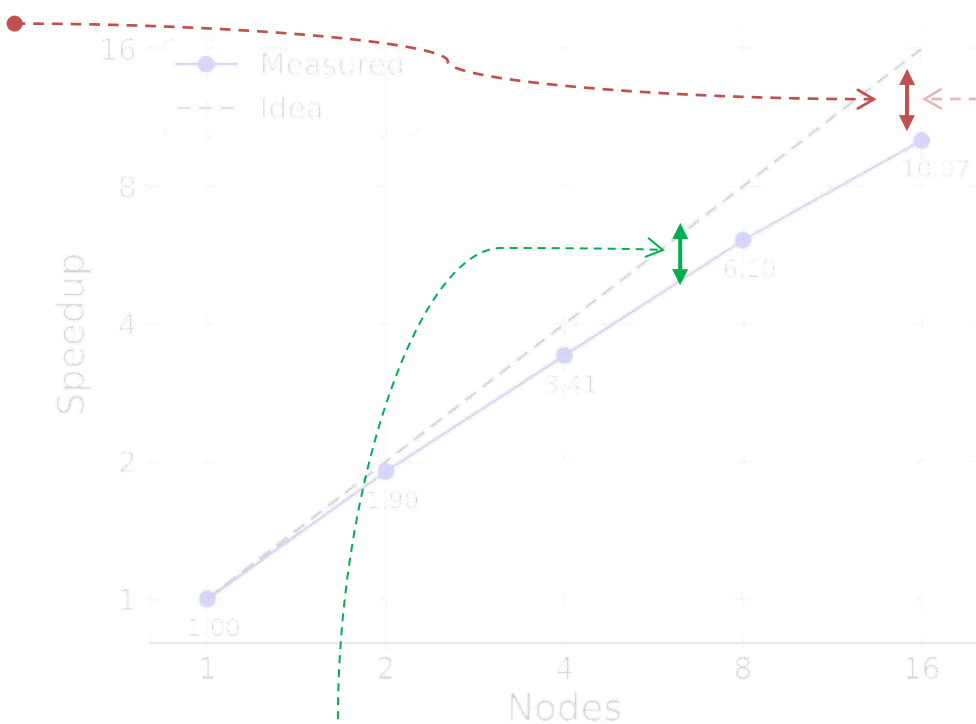
There will always be some loss of efficiency for a large scaling factor, because for a large-enough amount of resources the parallel overhead will be dominant. There is not a formal definition of this threshold, since that is very much dependent on the problem and the algorithm.

However I would say for good code that should happen beyond  $16\times \dots$

It may be worth to stress that the problem's size must also be appropriate to scaling.

If that is too small since the beginning, your parallel overhead will be immediately dominant.

Getting what is “small” or “adequate” is obviously part of the exercise..



gap due to inherent non-parallelism,  
communication, overhead due to inefficiency  
or to parallelization itself?



# | Weak Scaling

**Weak Scaling** measure how the time-to-solution varies when you grow the amount of resources  $n$  while keeping the workload per resource constant.

For instance, for the case under exam, the problem's size scales as  $N_{\text{grid}} \times N_{\text{grid}}$  and then the workload per resource is  $N_{\text{grid}}^2/n$ .

Then, when you enlarge  $n$ , you have to keep this ratio constant to maintain the workload.

Reporting the weak scaling correctly is easier than for the strong scaling, since the expectation is that the run-time remains constant. Then, also the absolute timing is “correct”. However, still, reporting the **efficiency** in this case give a better glance of how much the code looses (10%, 20%, 50% ?)

That`s all folks, have fun

“So long  
and thanks  
for all the fish”