NTNU

Norwegian University of Science and Technology

Anatomy of a Bulk-Synchronous Program

Presentation for RSE-team

11.11.2021

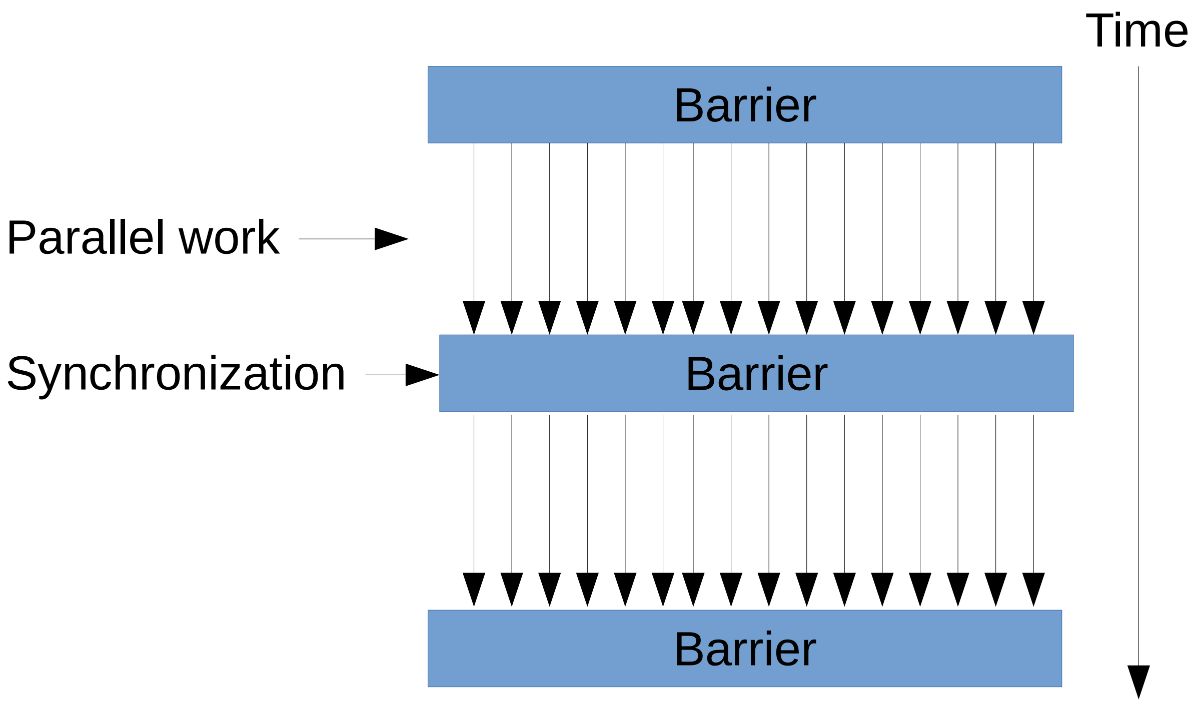
Jan Christian Meyer

What we’re doing today

- Architectural features only appear as statistics in software performance.
- The designs of memory systems and networks appear as *very clear* statistics in HPC program performance.
- When we last spoke, I described parts of why this is so.
- Today, we’re going to create a super-simple example program, to see where the connection comes from in practice.

Bulk-Synchronous Execution

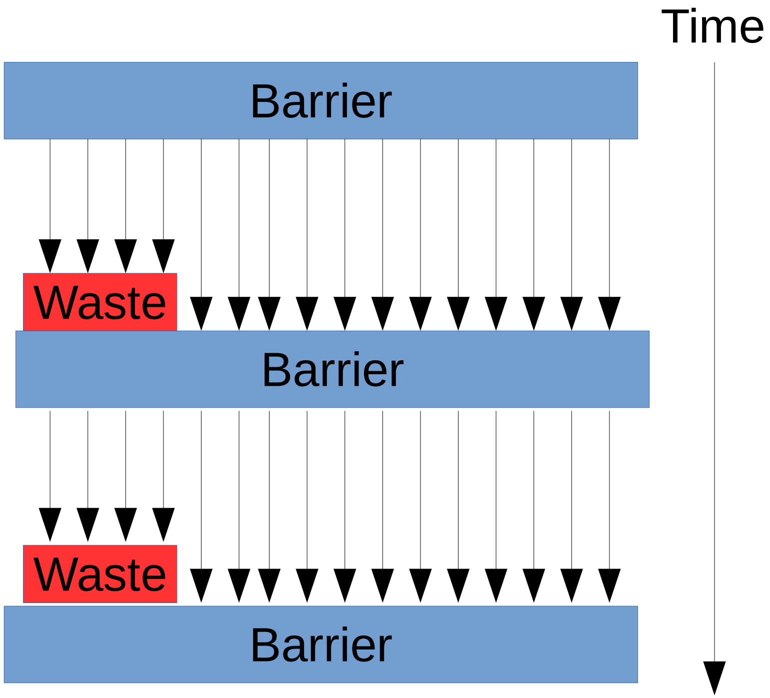
- This is a pattern that occurs in 75-90% of HPC applications.
- Superficially, it looks like this:



When some units are faster...

...they just have to wait, over and over.

- Corollary: *a supercomputer can only be as fast as its slowest component.*

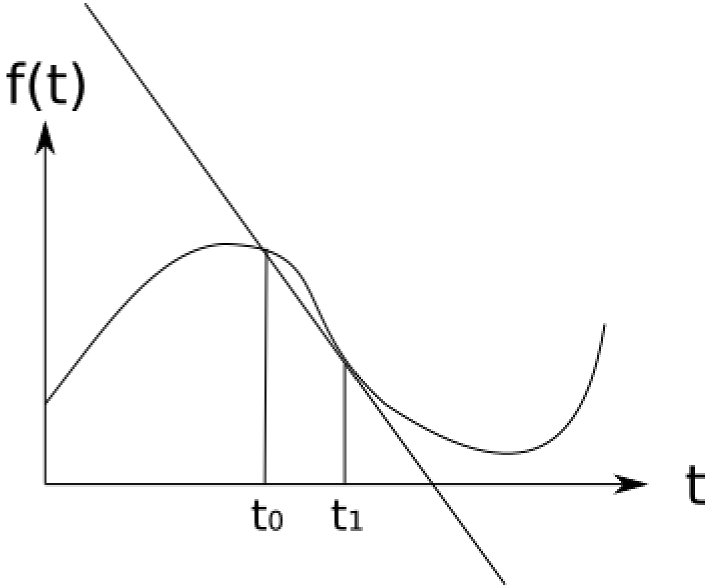


Where does this come from?

- Ultimately, it reflects the natural laws we simulate.
- Partly, it's because of the way we calculate approximations to them.
- We can try it out with a simple model of how heat disperses in various materials.

(There's more than conduction to heat transfer, but one equation is enough for now)

Derivative by Euler’s Method



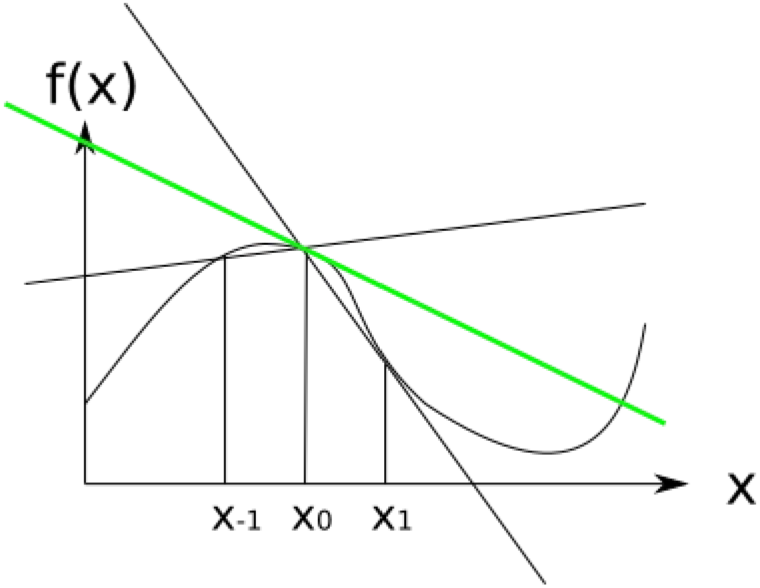
$$\Delta t = t_1 - t_0$$

$$f(t_0 + \Delta t) = f(t_1)$$

$$f(t_1) \approx f(t_0) + \Delta t \cdot \frac{\delta f}{\delta t}$$

$$\frac{\delta f}{\delta t} \approx \frac{f(t_{i+1}) - f(t_i)}{\Delta t}$$

Derivative by Central Difference



Estimate between -1 and 0 → $$\frac{\delta f}{\delta x} \approx \frac{f(x_0) - f(x_{-1})}{\Delta x}$$

Estimate between 0 and 1 → $$\frac{\delta f}{\delta x} \approx \frac{f(x_1) - f(x_0)}{\Delta x}$$

Their average → $$\frac{\delta f}{\delta x} \approx \frac{f(x_1) - f(x_{-1})}{2\Delta x}$$

2nd Derivative by Central Difference

Derivative from the right step

Derivative from the left step

$$\frac{\delta^2 f}{\delta x^2} \approx \frac{\frac{f(x_1)-f(x_0)}{\Delta x} - \frac{f(x_0)-f(x_{-1})}{\Delta x}}{\Delta x}$$

Differentiate those two

$$\frac{\delta^2 f}{\delta x^2} \approx \frac{f(x_1) - 2 \cdot f(x_0) + f(x_{-1})}{\Delta x^2}$$

Clean up a bit

How heat diffuses (in 2D + time)

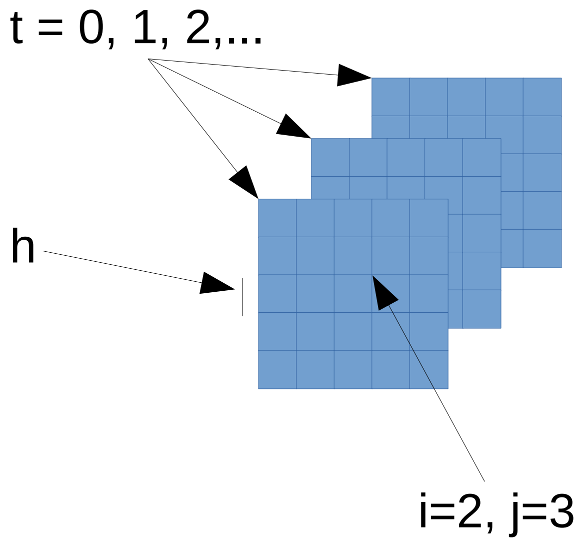
- Nature has it that

$$\frac{\delta u}{\delta t} = \alpha \cdot \left(\frac{\delta^2 u}{\delta y^2} + \frac{\delta^2 u}{\delta x^2} \right)$$

- For our discrete representation, say that

$$u(t, y_i, x_j) \Leftrightarrow u_{ij}^t$$

$$\Delta x = \Delta y = h$$



Substitute Our Approximations

$$\frac{\delta u}{\delta t} = \alpha \cdot \left(\frac{\delta^2 u}{\delta y^2} + \frac{\delta^2 u}{\delta x^2} \right)$$

becomes

$$\frac{u_{ij}^{t+1} - u_{ij}^{t}}{\Delta t} = \alpha \cdot \left(\frac{u_{i+1,j}^{t} - 2u_{ij}^{t} + u_{i-1,j}^{t}}{h^2} + \frac{u_{i,j+1}^{t} - 2u_{ij}^{t} + u_{i,j-1}^{t}}{h^2} \right)$$

Tidy up, and solve for next step in time:

$$u_{ij}^{t+1} = u_{ij}^{t} + \Delta t \cdot \alpha \cdot \left(\frac{u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} - 4u_{ij}^{t}}{h^2} \right)$$

Direct Translation to Code

$$u_{ij}^{t+1} = u_{ij}^{t} + \Delta t \cdot \alpha \cdot \left(\frac{u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} - 4u_{ij}^{t}}{h^2} \right)$$

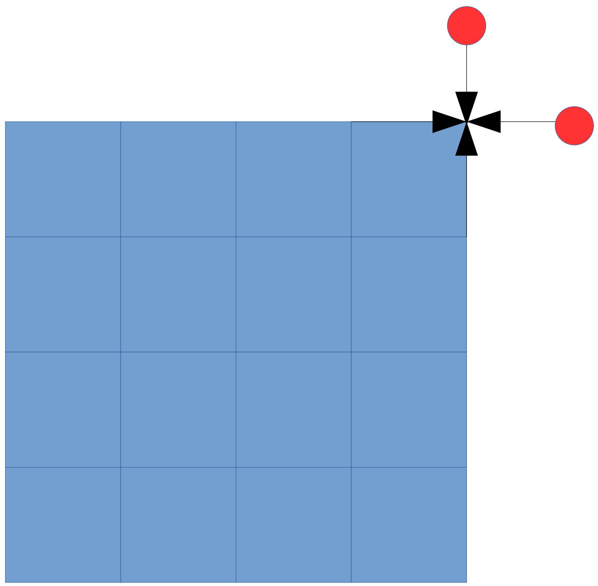
becomes

```
T_next(i,j) = T(i,j) + dt * (
    T(i+1,j) + T(i-1,j) + T(i,j+1) + T(i,j-1)
  - 4.0 * T(i,j)
) / (h*h);
```

if we let alpha = 1 for simplicity

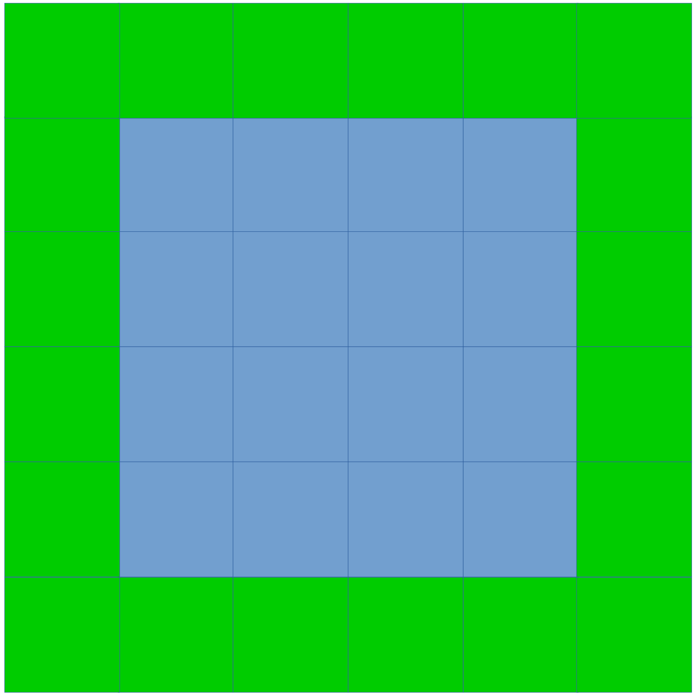
Boundary Conditions

- Each point requires values from its 4 neighbors
- All good things (and arrays) come to an end
- What can we do where two or more points are missing?



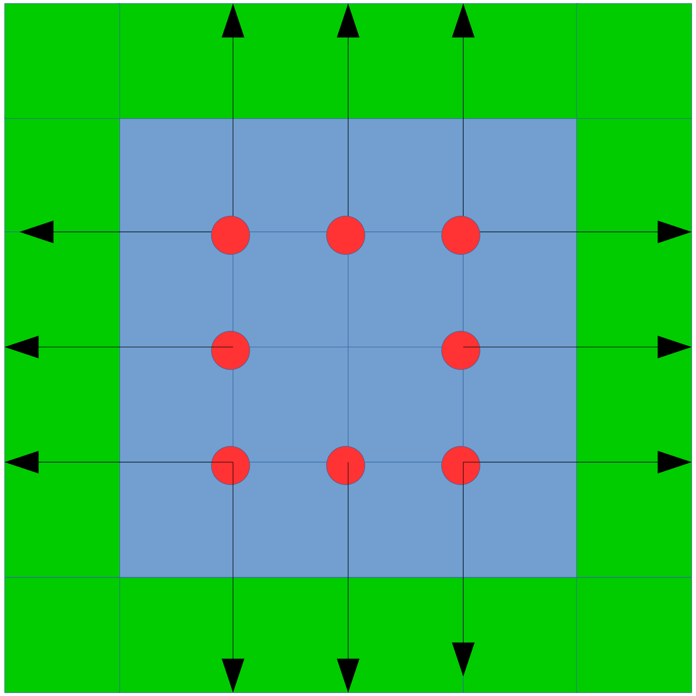
Make Something Up^{TM}!

- Dimension the array with padding on the sides
- Manipulate those values apart from the physics



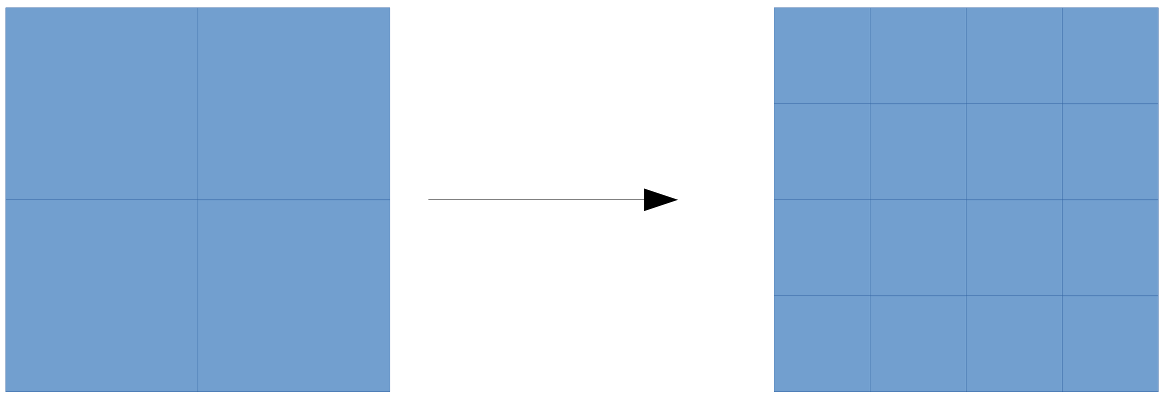
Neumann Boundary Condition

- Reflect values from inside the domain across the boundary
- This corresponds to saying that the derivative is 0 there:



Improved Resolution

- We can simulate the same thing again, with a more fine-grained grid.
- Let’s divide the cell edges in half, and get four times as many grid points.



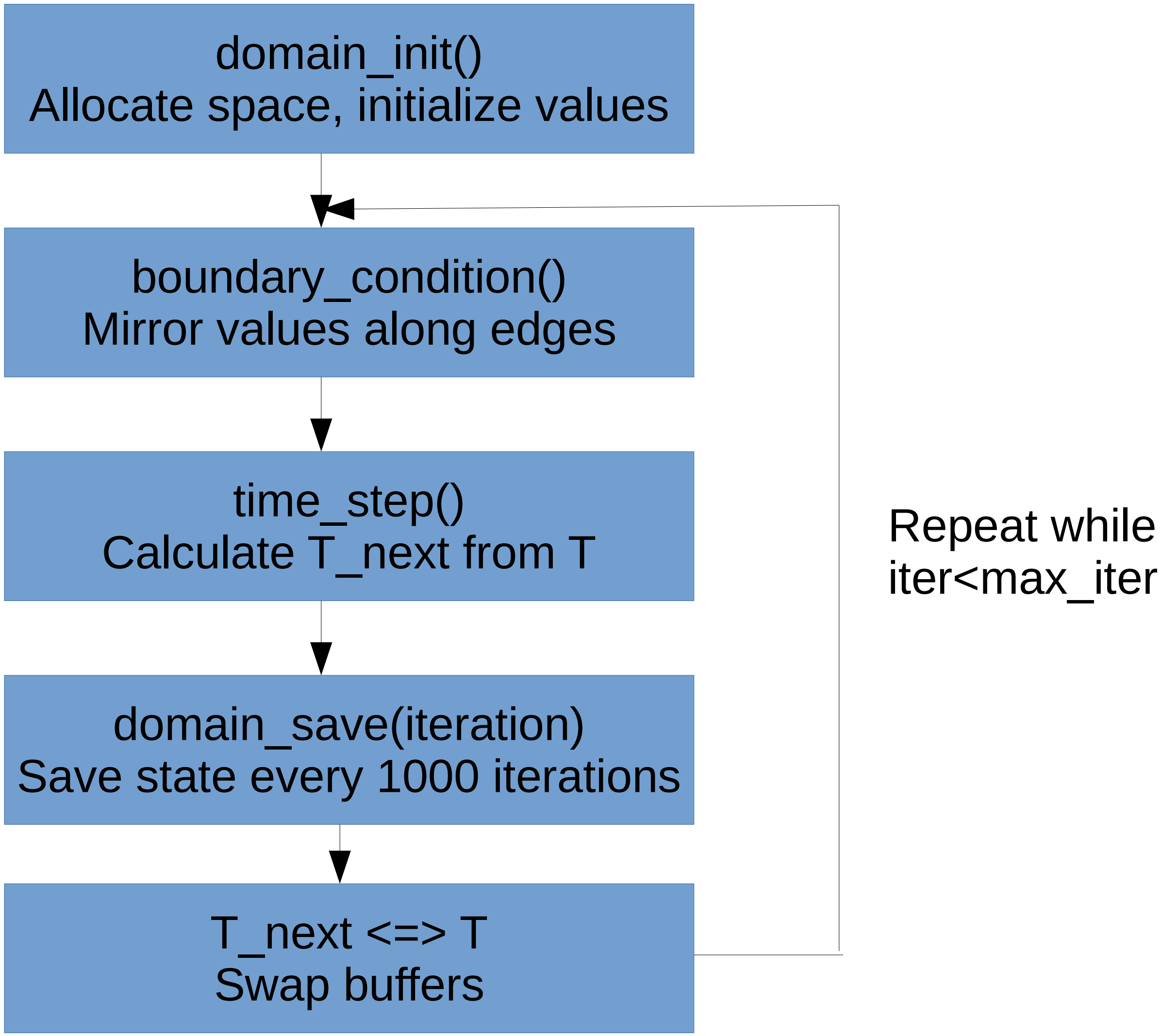
Impact on Simulated Time

- The gradients we estimate in space-dimensions are multiplied by the length of the time step when we integrate.
- If you take a small difference over one centimeter and multiply it by a million years, you'll get a number with no connection to reality.
- For numerical stability,

$$\Delta t \leq \frac{h^2}{2 \cdot \alpha}$$

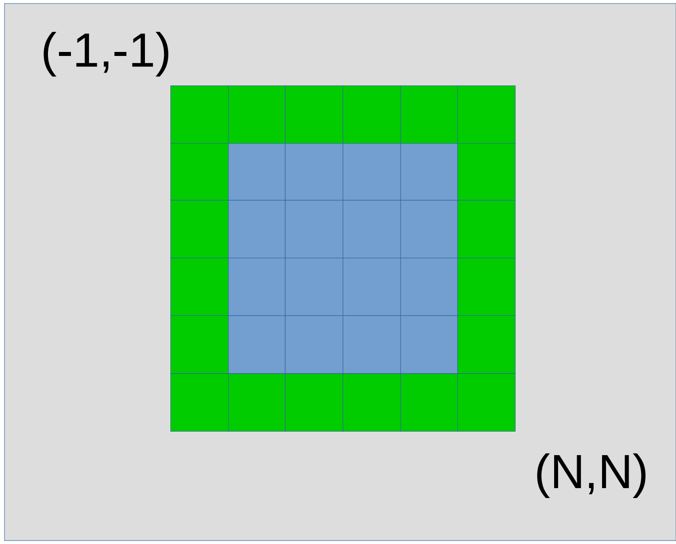
- Things can often go a bit wobbly even when they're equal, so I'll use 4 alphas in the denominator, to be on the safe side.

A sequential implementation



Indexing macros

- Buffers are allocated with (N+2)*(N+2) size, to have space for our halo of extra values
- `#define T(y,x) temperature[((y)+1)*(N+2)+(x)+1]` allows us to write T(-1,-1) and T(N,N) without causing segmentation faults
- This is just an indexing trick, but extremely helpful to keep things clear
- Also useful later on, with MPI



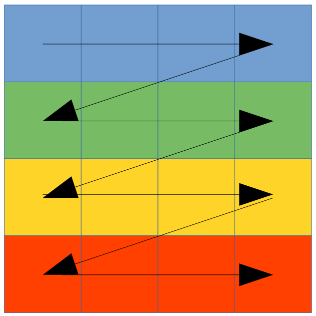
So far, so good

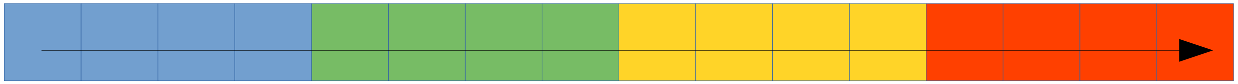
- Now that we have a working program, we can try things that affect its performance
- Without even going parallel, we can measure the effect of its cache utilization
- By multithreading the time_step() function, we can measure the impact of multicore cpus

...and see what happens if we create *false sharing*...

Cache Utilization

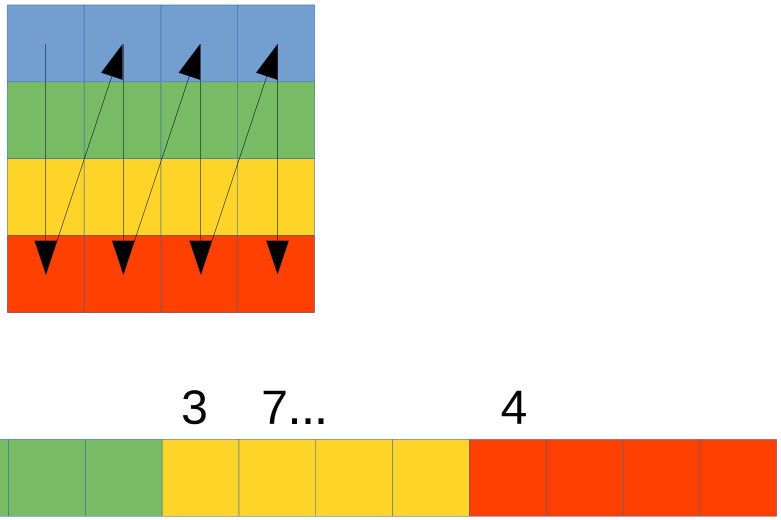
- Our program is not yet parallel, but we can already measure the impact of its memory access pattern.
- The arrays are laid out in memory by row-major ordering:





Cache Utilization

- If we traverse them in column-major order, we get an access pattern that is strided by the array size:
- There could be re-use in this order as well, but when the array grows big enough, the latest fetches begin to evict the first before the loop wraps around.

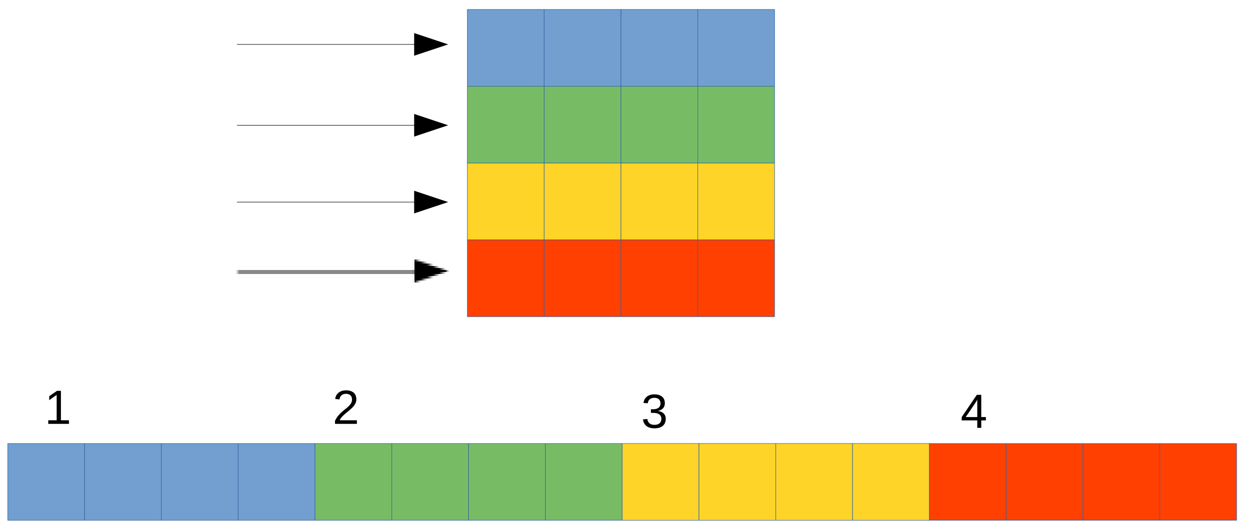


Multithreading

- Our time-steps must be sequential, but all the space-steps can be done simultaneously.
- This is a perfect case for applying the OpenMP programming model:
  - If you write
    #pragma omp parallel for
    in front of a loop, its iterations will be automatically distributed among threads.
  - The threads will join and vanish *after* the loop, so none of them speed on through to the next timestep.
  - This is one kind of *barrier* from the bulk-synchronous pattern.

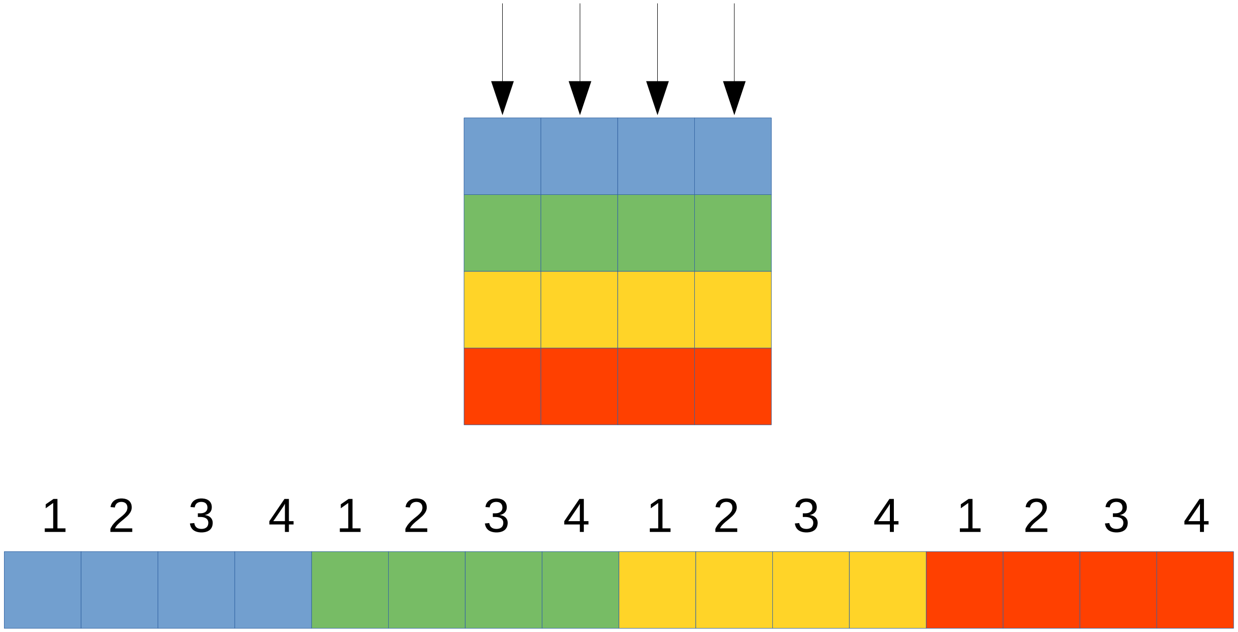
False Sharing

- If we distribute the work by rows, each thread gets a long, contiguous sequence to cache all by itself



False Sharing

- If we distribute it by columns, neighboring threads will cache values of interest to each other.
- When one writes to *its* location in the contested cache line, it will invalidate the other, even if there is no race condition.



That’s SMP in a nutshell

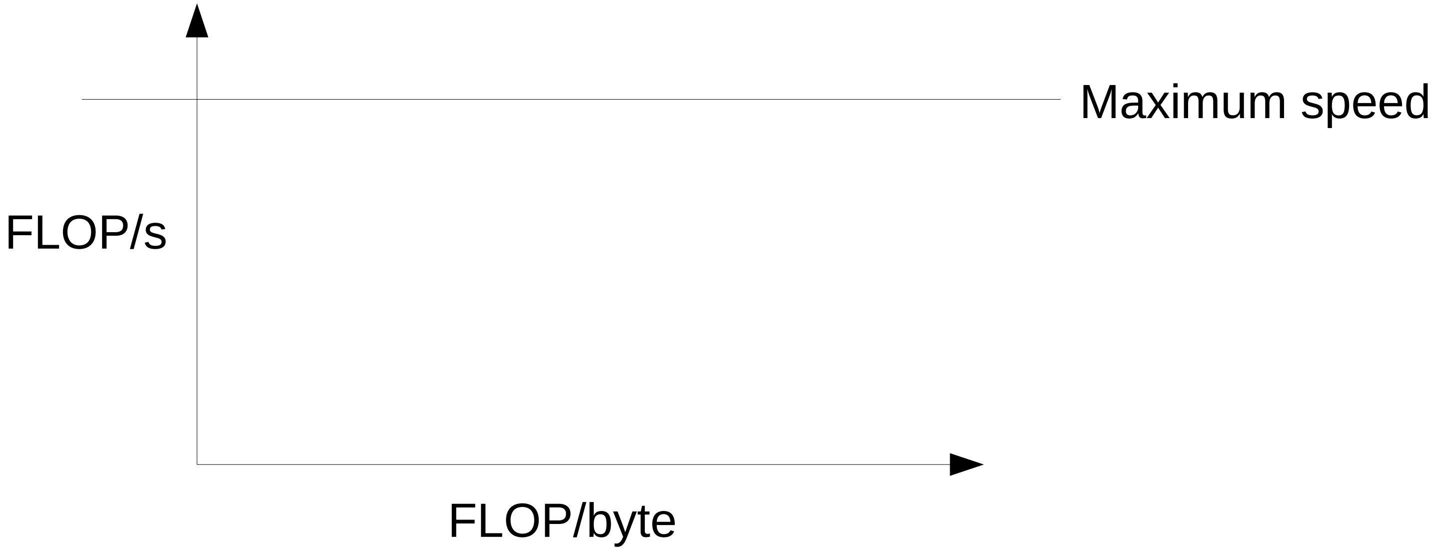
- We can now employ any size of shared-memory machine.
- This gets you into 4-digit core counts if you
  - rewrite it to use a graphics processor, or
  - spend 60.000.000 NOK on it.
- How fast can it go?

Roofline analysis

- Let the y-axis represent FLOP/s, and measure sustained computing rate, and
- let the x-axis represent how many FLOP-s the program carries out for each byte:

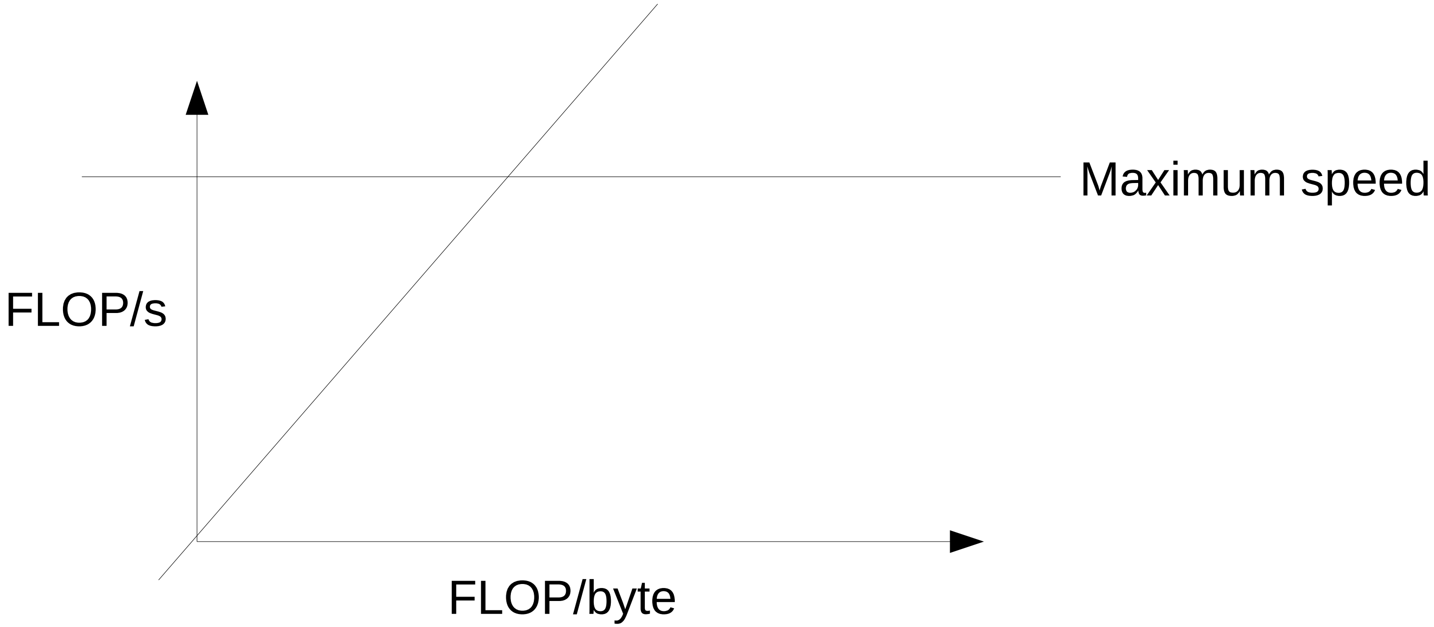
Peak computation rate

- If memory were as fast as the processor, the computer could calculate at its highest clock speed:



Peak memory bandwidth

- If the program only carries out a few operations per data element, it will be bottlenecked by the memory bandwidth.
  - [bytes/second] x [FLOP/byte] = [FLOP / second]



Our computation

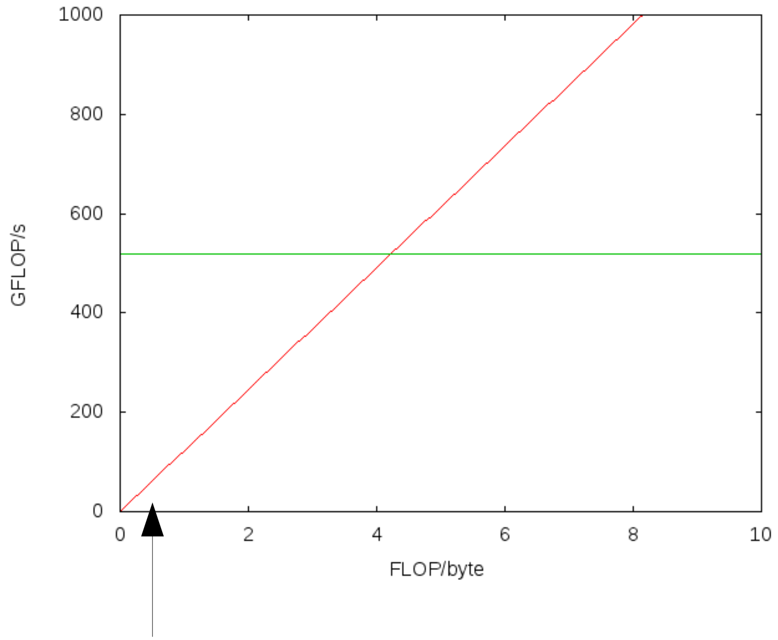
- The number of operations per datum is a characteristic of the computation, it's built in.

```
T_next(i,j) = T(i,j) + dt * (
    T(i+1,j) + T(i-1,j) + T(i,j+1) + T(i,j-1)
  - 4.0 * T(i,j)
) / (h*h);
```

- Ours has 10 operations for 7 8-byte values, that makes for an *operational intensity* around 0.1786.

Roofline conclusion

- Here’s a measured roofline graph from a 36-core Dell PE730 server:



We are apprx. here.

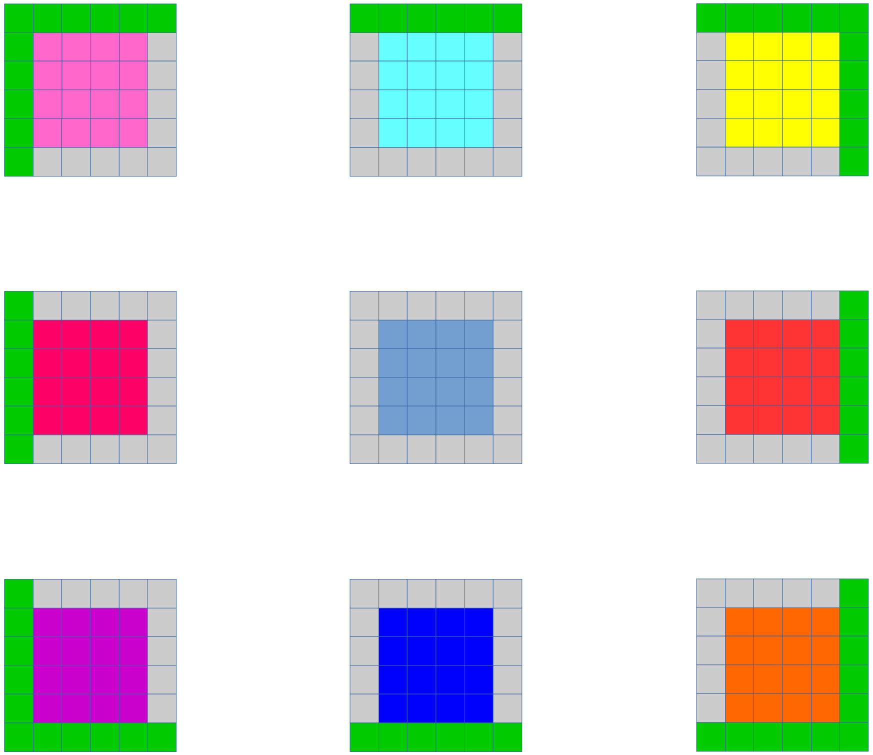
→ **This program will run at the speed of memory.**

Beyond threads

- When we run out of cores with shared memory, the next step is to use *distributed* memory
- This means we'll have to
  - launch separate copies of the program on separate computers,
  - put them in touch with each other, and
  - write them so that they figure out how to split the problem.
- Thankfully, MPI is here to help.

The issue we face

- If we split our 2D array into 9 separate pieces, here’s what we get:



First things first

- In order to get this grid of processes, the MPI implementation of our program begins by
  - counting the number of processes,

    `MPI_Comm_size`
  - configuring a «cartesian communicator» (*i.e.* a grid),

    `MPI_Dims_create`

    `MPI_Cart_create`
  - finding own coordinates in it,

    `MPI_Cart_coords`
  - and figuring out the east/west/north/south neighbors

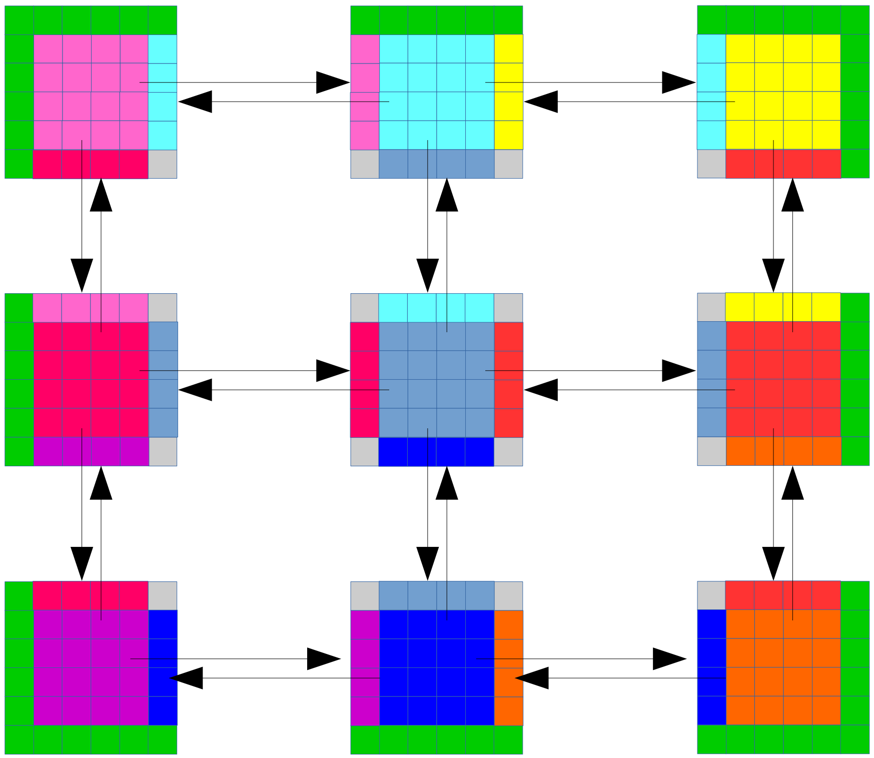
    `MPI_Cart_shift`

NTNU

Border exchange

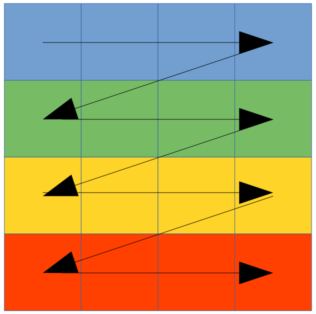
- In order to calculate the values for each point that is adjacent to the (grey) halo points, we must fetch its value from the neighboring process

Border exchange, illustrated



Memory layout

- As we already mentioned, the array is stored in row-major order:



- That means column vectors are strided in memory, such as the leftmost one here, which will occupy indices {0,4,8,12}

NTNU

Data types for communication

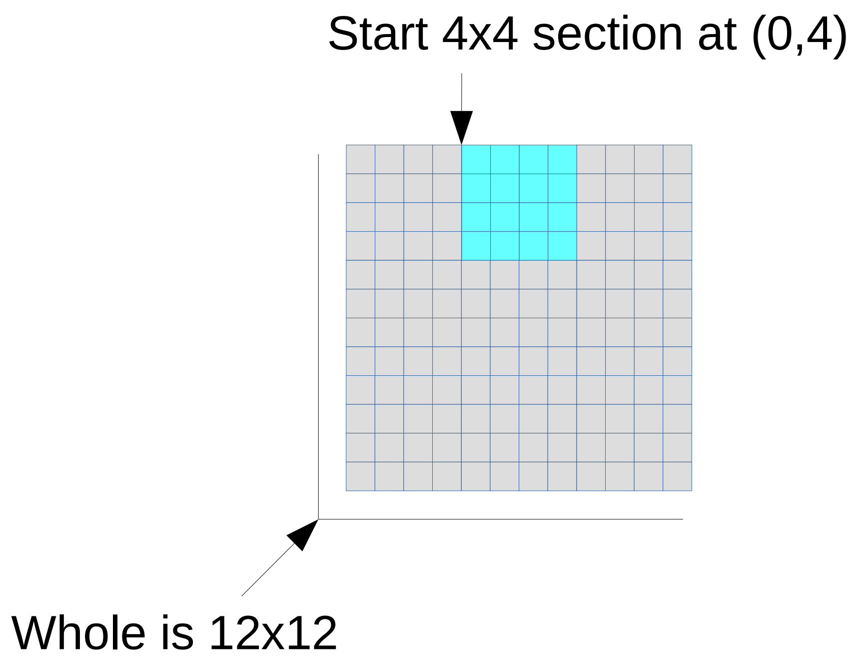
- MPI can store memory access patterns with gaps in, to make such things easier to handle
- In the function `setup_mpi_types`, the two calls to `MPI_Type_vector` create a row and a column vector type for the border exchange
- The `border_exchange` function uses them to swap values between neighbors according to the diagram

Saving results

- We have a related issue when saving the entire array to file.
- Files are (ostensibly) sequential, but the distributed array isn't.
- MPI datatypes can also express addressing of rectangular slices from a whole, `MPI_Type_create_subarray` requires
  - The size of the whole it is indexing into,
  - the size of the slice it is supposed to index, and
  - the coordinates of the slice's starting point/origin.

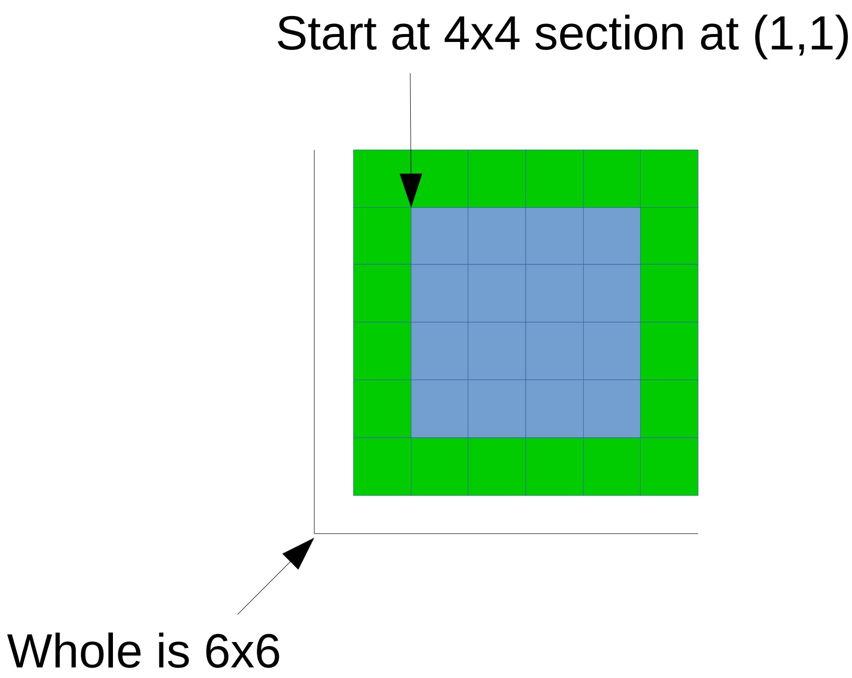
We need 2 of these

- One for indexing where to *write* values (called `domain`), *e.g.* for the top/center process:



We need 2 of these

- One for indexing where to *read* values (called `subdomain`):



Parallel I/O

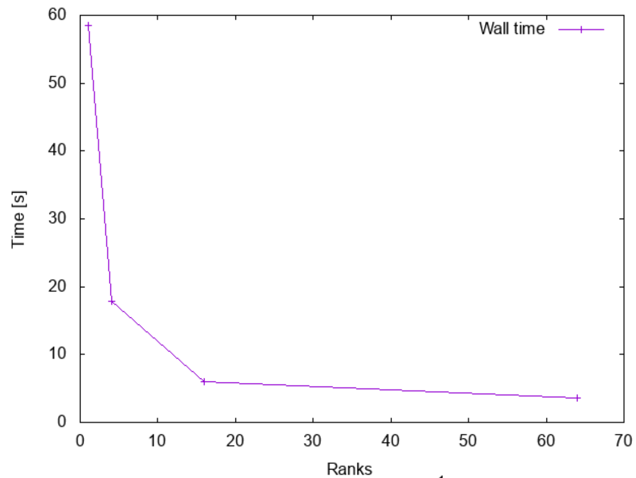
- Armed with these datatypes (also configured in setup_mpi_types), we have parallelized I/O as well

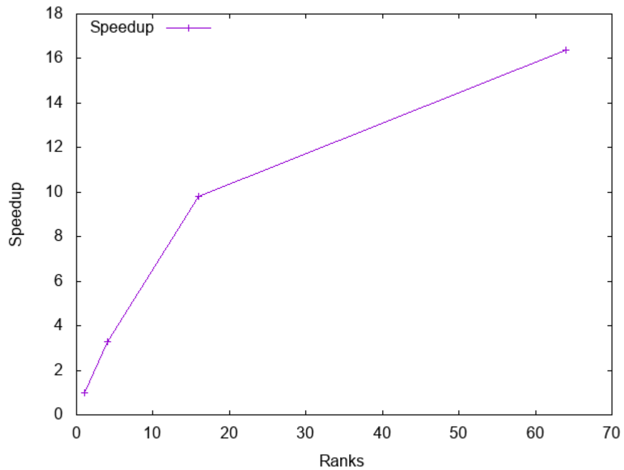
  (...as long as the file system supports it...)

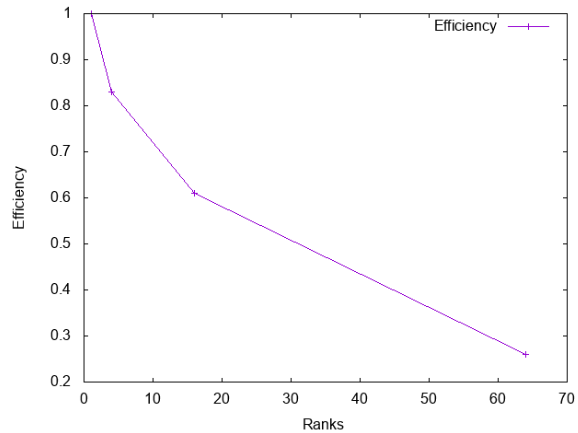
- Our program is now «entirely parallel»
  - No kings, no masters
  - *(...it still has to launch and stop, though...)*
- So, what kind of performance can we get?

Strong scaling results

- With a problem size of 512x512 points:







NTNU

We run out of work

- At 8x8 ranks, each subdomain is only 64x64 points
- Additional ranks contribute little, we've reached diminishing returns (*cf.* efficiency curve)
- Still, we cut execution time by a factor 16.4

Is it worth the trouble?

- What it cost us
  - We've used hardware that costs roughly 260.000 NOK
  - Code has almost doubled in size (whatever that costs)
- What we gained
  - 16-17 times faster execution for *this* problem size
  - Ability to finish (almost) arbitrarily much larger problems, in exchange for additional hardware
- Conclusion: «it depends»
  - Consider the problem you want to parallelize

How realistic is the comparison?

- Just for fun, I also wrote up the exact same program logic in Python (3)
  - Numpy arrays for slight speed improvement
  - Direct translation, so it's not adapted to Python-isms

    (...please show me any improvements you know of...)
- Short and pleasant exercise, 60% code reduction for sequential version
- Measured wall time on same hardware: **74797.81s** (20.78 hours)
  - It took longer to run this version than to write all variants put together
  - If we could get 16x-17x-ish speedup, it would still run for 1h15m

Thank you for your attention!

Are there any questions?