Short:

- Talk about my team's contribution to EC's Destination Earth project
- GPU Research Engineer at BSC
- Performance team at BSC under earth sciences department
- Links to social media

Long

Welcome to this talk where I'll be talking about my team's contributions to the European Commission's Destination Earth project.
Me name is Okke van Eck, I'm a research engineer at the Barcelona Supercomputing Center, short BSC.
I'm specialized in GPU programming, but do performance engineering in general.
Here you can find my email address and links to my social media in case you want to ask some questions afterwards.

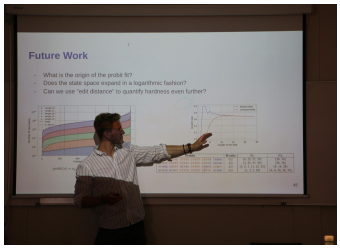
Who's Okke?

Universiteit van Amsterdam

via

VU Vrije Universiteit Amsterdam

Barcelona Supercomputing Center, Centro Nacional de Supercomputación



Short

- Bachelor CS at UvA and board 19 at VIA.
    - SNIC advisory board in 2019-2020
- Master's PDCS at VU
    - Awarded the Unilever Research Prize 2023
- Moved to Barcelona for BSC
    - Here small part of the team in the beginning
- Now we have a much bigger team, mostly consisting of juniors
    - Having a lot of fun, for example dinners together and go rock climbing on the weekends

Climate is changing..

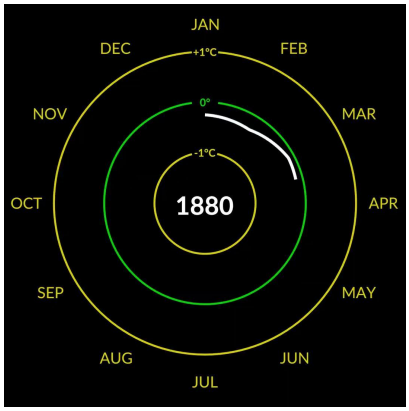
The core problem we analyze is climate change.
I want to show you how bad it really was last year in 2023.
This video from the Guardian (news Australia) highlights some of the effects of climate change.

Climate is changing..

VIDEO PLAYS AUTOMATICALLY

https://www.youtube.com/watch?v=pf4aOIE_3bc

Climate is changing..

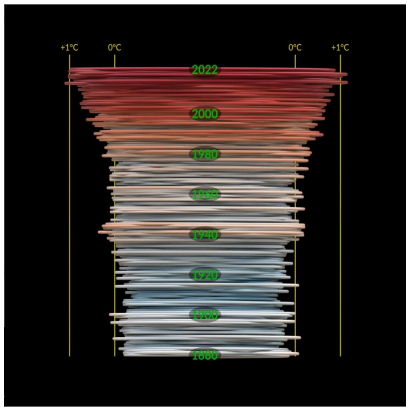


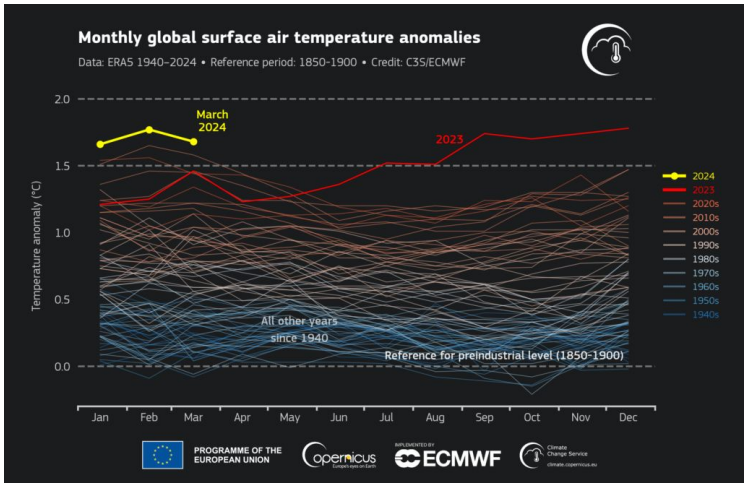
VIDEO PLAYS AUTOMATICALLY

So as you saw, the effects of climate change are very real and very destructive.
To make it even worse, this is just the beginning and the problems will get way worse.

In this circle, you see the temperature difference per month relative to the average temperature of 1900-2000 (20th century).
As you can see we have first colder moments.
Then there is a small increase at 1945 due to the 2nd world war.
But from 1980 the temperature really starts to rise.
And from 2019 it got even worse.

Climate is changing..

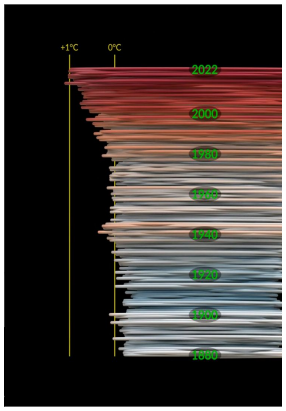


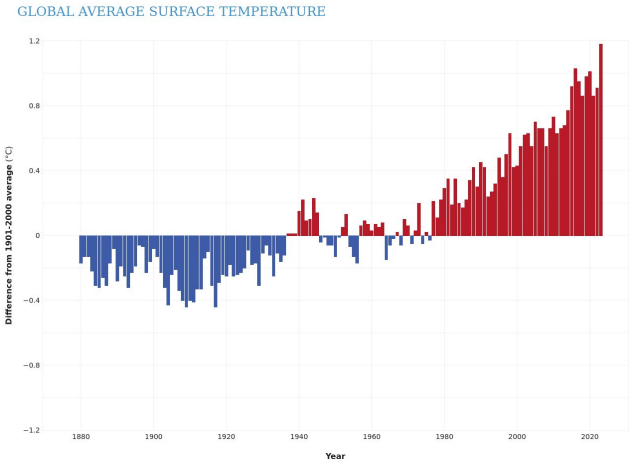


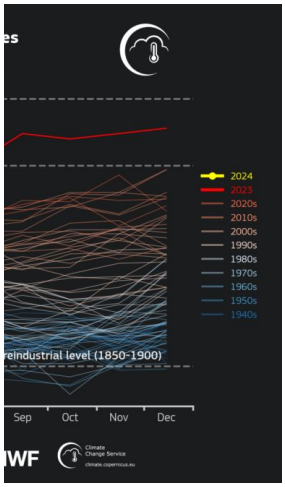
Here you can see the spiral in full, and it's clear that it's getting incrementally worse from the 2000's onwards.

On the right side, you can see it in another view from last march where 2023 and 2024 were clearly breaking previous records.
Look at the gap in the summer between 2023 and the rest, we are really accelerating now.
And unfortunately, 2024 broke the records of 2023 afterwards.

Climate is changing..







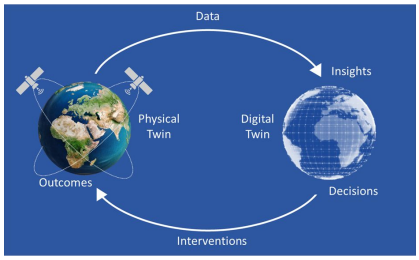
So things are getting a lot worse lately.
Here you have a bar-chart version of the same data.
There is one bar missing for 2024, which will be higher than 2023.

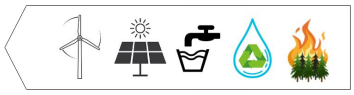
Destination Earth: Digital Twins

And so, clearly, something needs to be done.
I hope you are not feeling too hopeless, but this is reality, and we achieve nothing by sitting on our hands.

Thus the European Commision has started the Destination Earth project in 2022, where they aim to build a digital twin of the earth.

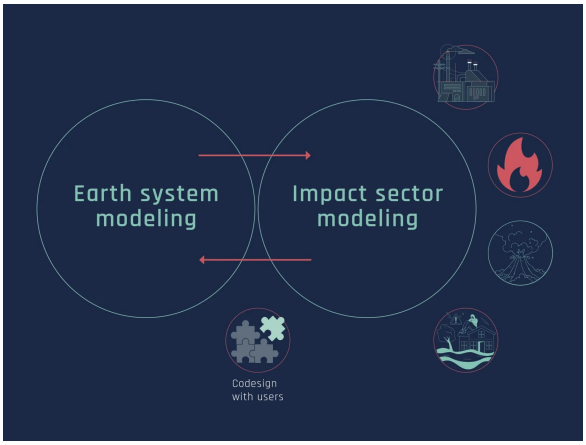
Why digital twins?





Impact sector applications

esa



Let me explain what a digital twin exactly is.

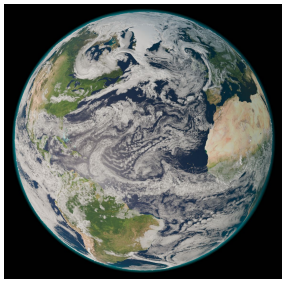
Here you can see the basic premise.
We have a physical twin, the earth, where we can measure current activity with the help of satellites.
Then we want to use that data to simulate what our climate will do in the future.
This is done in the digital twin, which is a model of the earth.
Based on the predictions of this model, we can intervene and change policies to better our climate.
These policies have an effect, which we can measure again through satellites, feed to the model, and repeat.

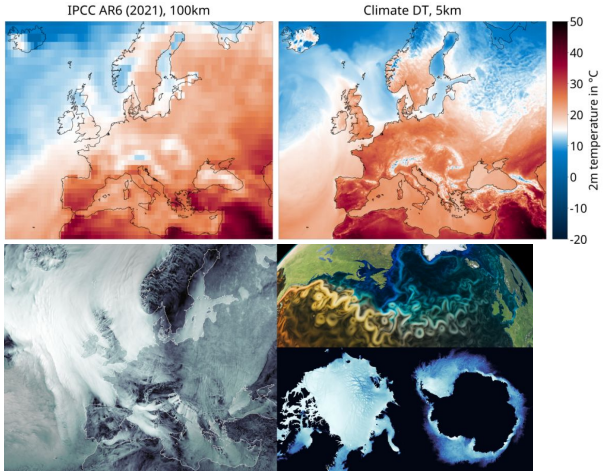
This idea of using a digital twin consists of 2 sectors.
We have Earth System Modelling, which models the physics of the earth.
Then you have Impact Sector Modelling, which models the effects of the climate on specific sectors.
For example the effect on forest fires or floods.

These impact sector applications are viewed by ESA to advice policy makers on potential interventions.
We will see this later when I'll explain how we run a simulation.

Destination Earth: Digital Twins

- Phase 1 ran from 2022-2024
- 3 different models
- One simulation takes months
- 300+ supercomputer nodes
- Multiple EuroHPC supercomputers





Thus we have the Destination Earth project, where the Digital Twins are a subproject of.
The goal is to create a highlight accurate digital twin at global scale,
which monitors, simulates, and predicts the intersection between natural phenomena and human activities
Phase 1 finished last april and took 2 years.
There are 3 or 4 other phases of 2 years coming up, so in total it will be a project till +-2030.
In this first phase, we were able to achieve a 5km resolution as you can see in the left upper corner.
This is much more detailed than the 100km standard from before the project.
Russia is actually cold now for example.

One important thing is that we do not use 1 model, but 3 different ones.
Each model has a bit of a bias towards some part of the physics.
It is super complex to get it exactly right, because the laws of physics are incredibly complex.
Instead of optimizing 1 model, we have 3 and take the average of them for the predictions.

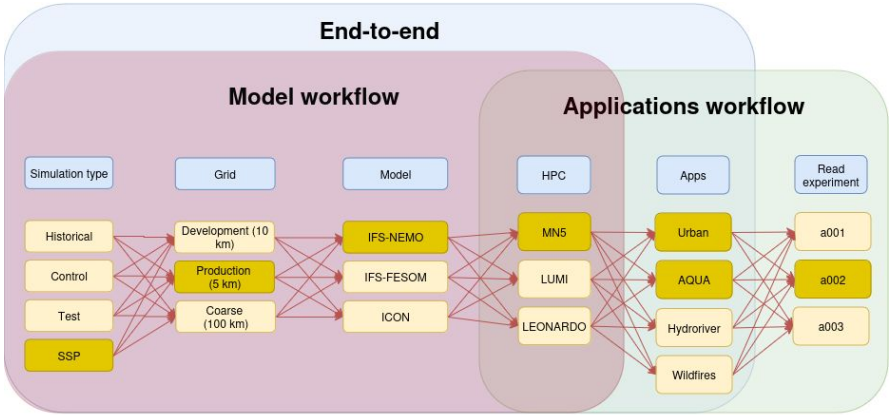
One of these simulations takes multiple months for a 50 year prediction.
It requires 300+ supercomputing nodes to achieve this simulation speed.
As you can imagine, this takes a lot of power as well.
Lastly, we have to develop the models to run on 3 different EuroHPC supercomputers.

Each have different architectures so the portability becomes a bit of a problem, but about that later more.

But performance is critical, so we tune the models to work as best as possible.

What does it take to run a model?

- AUTOSUBMIT



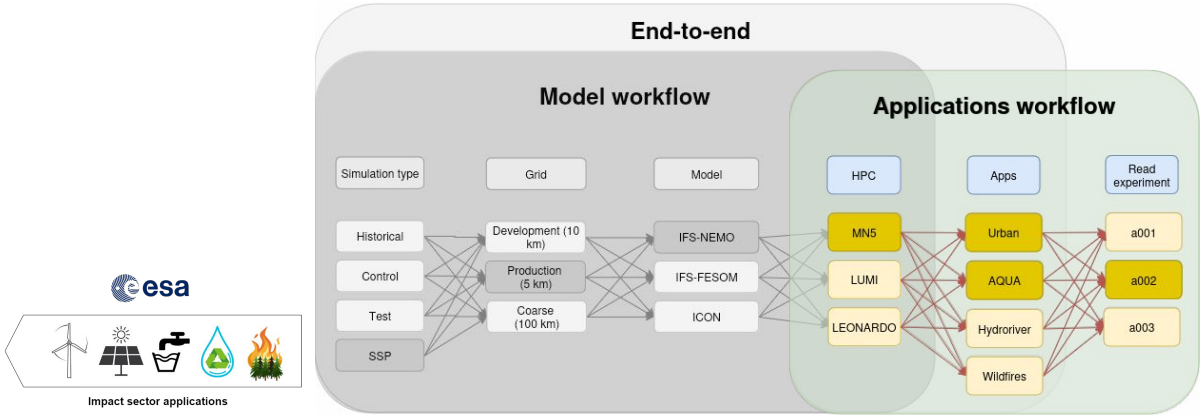
And so you might wonder, what does it take to run a model?
Well, for sure you will need a workflow manager.

AutoSubmit is our in-house open-source workflow manager, that is specially created for running climate simulations.
It gives scientists options for preparing, launching, and analyzing climate simulations.
As you can see here, it has different configurations we can choose from, depending on the simulation we want to run.
There is the "Model Workflow" and the "Applications Workflow".

SSP = Shared Socioeconomic Pathways
https://www.carbonbrief.org/explainer-how-shared-socioeconomic-pathways-explore-future-climate-change/

What does it take to run a model?

- AUTOSUBMIT

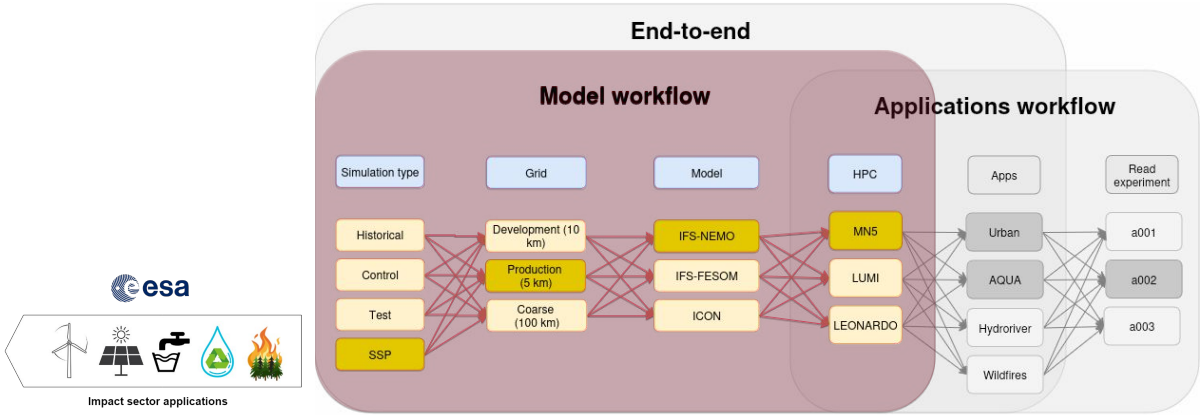


Let's focus on the application workflow first.
Here you have three parts, of which the first one is to select a high-performance computer (HPC).
Then on the 2nd column, we have different applications that analyze the experiments.

These correspond to the impact sector applications that we have seen before.
All of these need to read data from experiments, which is the last column.
So this is really easy. Load data, run the applications on HPC, and have a climate researcher look at the data.

What does it take to run a model?

- AUTOSUBMIT



Then we can also look at the Model Workflow.
This is the part where we actually start up simulations.

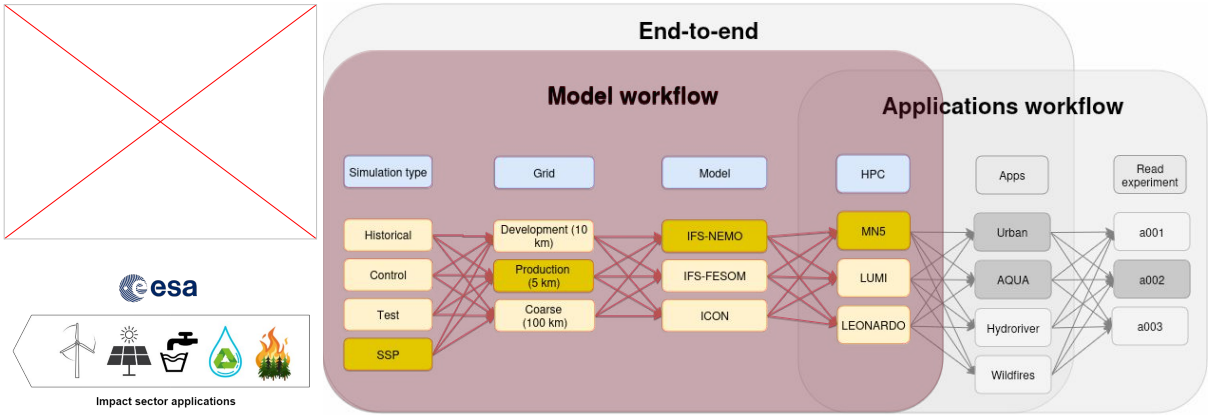
In the first column we select the simulation type that we want to do.
The meaning of SSP doesn't matter much for this talk, but it's basically some socio-economic situations.

Then in the 2nd column we select the grid size of the model, like we saw before.
We typically run with a 10km resolution for development and official 5km during simulations
NEXT SLIDE

SSP = Shared Socioeconomic Pathways
https://www.carbonbrief.org/explainer-how-shared-socioeconomic-pathways-explore-future-climate-change/

What does it take to run a model?

- AUTOSUBMIT

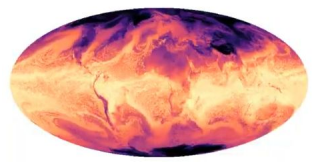


VIDEO PLAYS AUTOMATICALLY

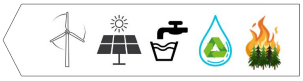
In this video you can see how the different grids are relative to each other.
This is managed through something called HEALPix.
It's a pixelation algorithm for sphere-like geometry, where each pixel always covers the same amount of area.

What does it take to run a model?

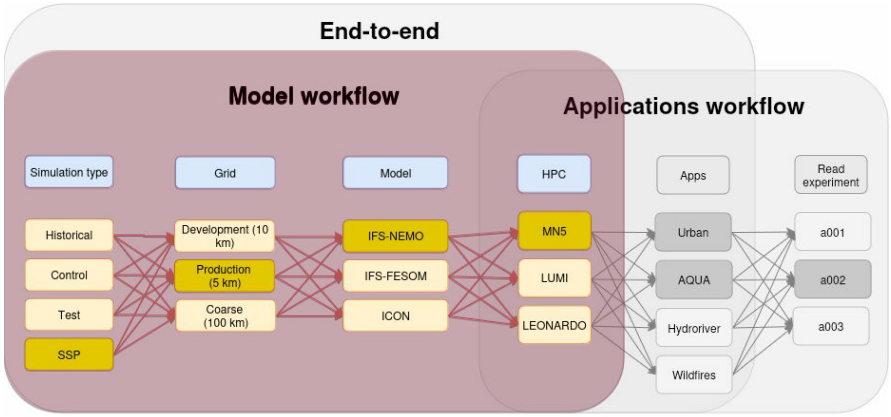
- AUTOSUBMIT



esa



Impact sector applications



Then in the 3rd column we select what model to run.
Like I said before, we have 3 models that we develop for averaging out biases.

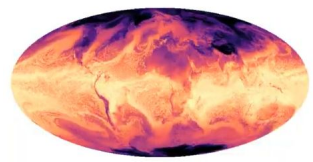
IFS is an atmosphere model, which is coupled with a sea model.
In case of the first 2 options, the sea models are called NEMO and FESOM.
Hence you can imagine how difficult the sea is to model, especially the sea ice.

Lastly we have ICON, which is a singular model which combines an atmosphere and a sea part.

Then in the 4th column we have the different supercomputers that we have to develop for, which I'll cover later.

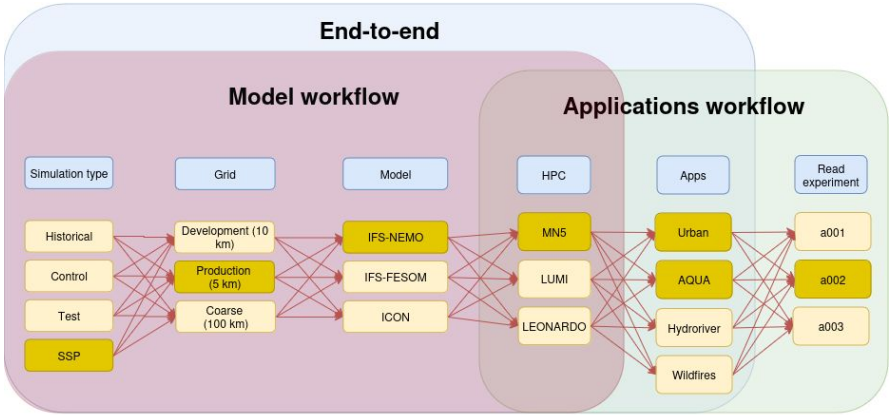
What does it take to run a model?

- AUTOSUBMIT



esa

Impact sector applications



Going back to the full picture, this is basically what is required for running a climate model.
Note the "End-to-End" workflow in blue, which basically forwards the output data straight to the apps, which eliminates the storage.

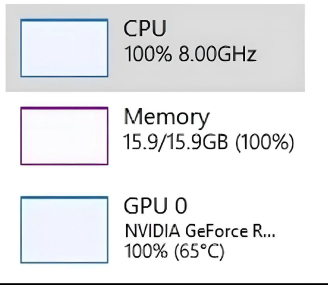
And thus if you combine everything together, you will be able to simulate the earth on different resolutions.
Here you can see the different resolutions for the sea part.

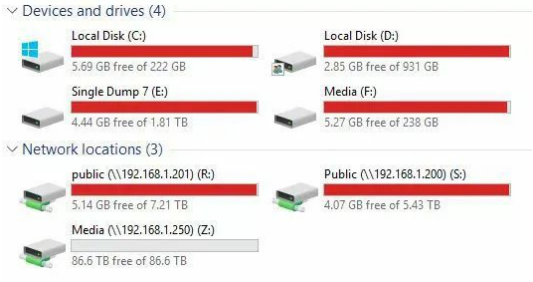
Performance Engineering

That sums up my introduction to climate modelling, and we can talk about what my team does.
As the Performance team, we look at models and try to understand inefficiencies that we can improve upon.
Climate models are generally written by climate scientists, and thus their performance is not really optimal for the modern day systems.
Let me explain a bit what performance engineering is about.

Performance Engineering

"I paid for the whole PC, I'm gonna use the whole PC"





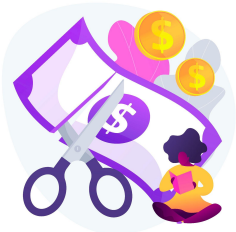
So, what is performance engineering?

One might think that performance engineering is about fully utilizing a system.

However, that's not really the case.
More optimally utilizing the system is our solution to some problems organizations face.

Performance Engineering

"I paid for the whole PC, I'm gonna use the whole PC"



Which are for example to reduce expenses.
Hardware is expensive, and thus buying extra machines to scale up is not always the best choice.
Maybe you can compute more with the hardware you already have.

Next, it's about sustainability.
By optimizing models, we reduce their compute time and thus we need less hardware to begin with.

Lastly, it's also about cutting CO2 emissions.
Reducing runtime makes supercomputers consume less energy and thus the emission is reduced.

Credits for the 3 icons:
https://www.freepik.com/free-vector/cut-price-bargain-offering-reduced-cost-discount-low-rate-special-promo-scissors-dividing-banknote-crisis-bankruptcy-cheapness-market-vector-isolated-concept-metaphor-illustration_12470246.htm#fromView=search&page=1&position=0&uuid=2212521e-e19b-4c7c-8e37-8f20c0c2fb4d

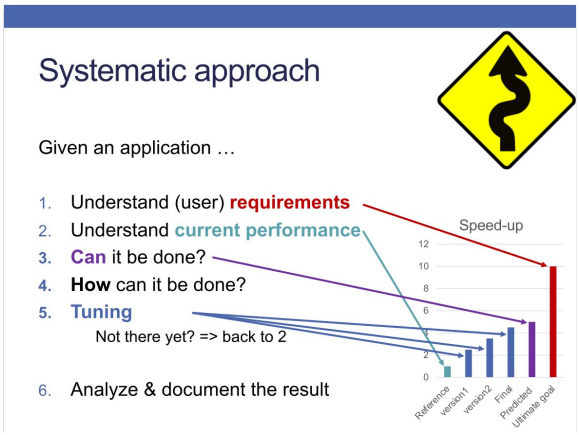
https://www.freepik.com/free-vector/resources-protection-abstract-concept-illustration-protection-natural-resources-land-conservation-safeguarding-nature-smart-water-use-environment-preservation_12145555.htm#fromView=search&page=1&position=0&uui

d=2d973d43-fda8-453f-8df7-9bd3a2177589

https://www.freepik.com/free-vector/stop-air-pollution-carbon-dioxide-reduction-environmental-damage-atmosphere-protection-toxic-emission-problem-ecology-volunteer-cartoon-character_12145539.htm#fromView=search&page=1&position=2&uuid=18f7ce33-9c1b-4904-a098-a0a807c4d5fb

Performance Engineering

Follow these simple steps:



So how does the process of performance engineering look like?
Follow these simple steps:

1. Understand the requirements that the user is asking for.
   Usually this is too ambitious.
2. Understand what we currently can do.
3. Predict what performance is possible.
4. Think of techniques to achieve this predicted performance.
5. Improve bit by bit and compare your versions!
   Can we actually achieve the prediction?
6. Analyze your final version and document why what was possible

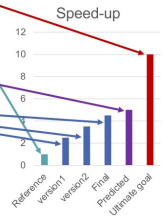
Performance Engineering

Follow these simple steps:

Systematic approach

Given an application …

1. Understand (user) **requirements**
2. Understand **current performance**
3. **Can** it be done?
4. **How** can it be done?
5. **Tuning**
   Not there yet? => back to 2
6. Analyze & document the result



Thank you Ana Varbanescu!

If these slides looked a bit familiar, I stole them from Ana Varbanescu.
She gave me my performance engineering course at the UvA and VU, and this slide is from her slide deck.
Thank you Ana for the great course (and slides)!
Now she mainly works at the TU Twente I believe, highly recommend following her courses.

Performance Engineering

Always optimize your application for a specific system!

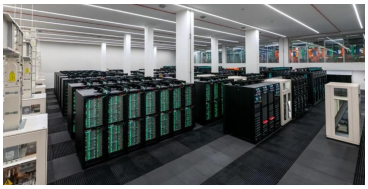
One thing Ana taught us, was to always optimize your application for a specific system.
This is due to caching behavior, network behavior, etc.

Performance Engineering

Always optimize your application for a specific system!

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MN5
MARENOSTRUM



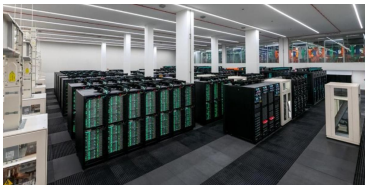
Well, then you get a European project, which doesn't ask you to optimize for one system..

Performance Engineering

Always optimize your application for a specific system!

Barcelona Supercomputing Center, Centro Nacional de Supercomputación

MN5 MARENOSTRUM



CSC

LUMI

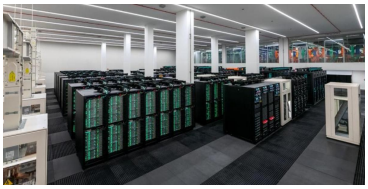
Not two systems..

Performance Engineering

Always optimize your application for a specific system!

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MN5 MARENOSTRUM

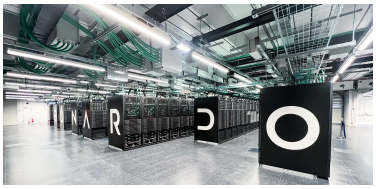


CSC

LUMI

CINECA

LEONARDO



But three different systems!
And the model should be optimized for all.

On the left you see MareNostrum 5, our supercomputer in Barcelona, Spain.
Managed by Barcelona Supercomputing Center
Then in the middle you see LUMI, which is the supercomputer in Kajaani, Finland.
Managed by CSC.
And on the right Leonardo, which is the supercomputer of Bologna, Italy.
Managed by Cineca.

Performance Engineering

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	Power (kW)	Energy Efficiency (GFlops/watts)
12	5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	7,107	53.428
15	8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR, EVIDEN EuroHPC/BSC Spain	663,040	175.30	4,159	48.320
28	7	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	7,494	32.187

These 3 computers are the most powerful in Europe, and also very green!
Here are their ranks according to the Green500 and Top500, which are world wide rankings.
You can see that they are the numbers 5, 8, and 7 world wide.

These are all partitions which have GPUs available, which make them more energy efficient than the CPU partitions.

Performance Engineering

Performed optimizations:

- CPU
- GPU

So we used these machines to perform some optimizations on the IFS-NEMO climate model, and I'd like to share some outcomes of those.
The problem with the optimizations is that they are quite technical and thus I don't think I can fully explain them within 45 minutes.
Hence, I will briefly mention them and continue explaining how we detect inefficiencies within code.

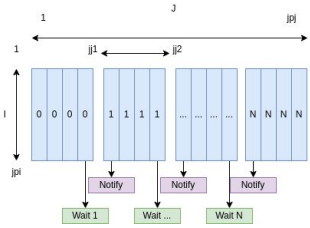
CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

Performance Engineering

Performed optimizations:

- CPU
  - OpenMP regions (1.2x - 2.8x depending on the region)
- GPU



First for the CPU optimizations, we have OpenMP regions that are optimized. Basically, OpenMP is a multiprocessing paradigm which allows for shared memory programming.
Sometimes, these processes have to wait for each other, creating time in which nothing is computed.
We optimized some of these regions and got a 1.2x to 2.8x improvement depending on the region.
Note that this is a REGIONAL improvement, not a GLOBAL one!
The global improvement is much less, typically we are happy with a 5% global improvement, which is 1.05x.

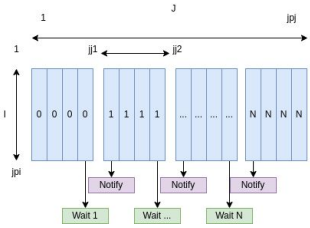
CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

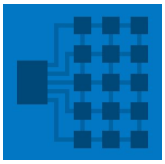
GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

Performance Engineering

Performed optimizations:

- CPU
  - OpenMP regions (1.2x - 2.8x depending on the region)
  - MultIO improvement (1.03x)
- GPU





Next, we did an improvement within the MultIO part, which resulted in a 1.03x GLOBAL improvement.

Might seem little, but is actually significant.

MultIO handles the parallel input/output of the model, which allows for writing results in parallel to computation.

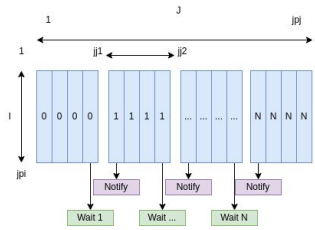
CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

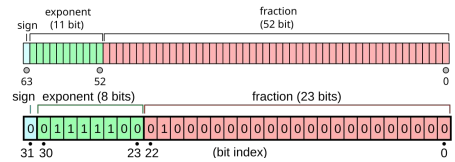
GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

Performance Engineering

Performed optimizations:

- CPU
  - OpenMP regions (1.2x - 2.8x depending on the region)
  - MultIO improvement (1.03x)
  - Mixed Precision (1.2x)
- GPU





Then we also introduced mixed precision to the model, which resulted in a 1.2x GLOBAL speedup.
Within computing you have single and double precision, which are 32 and 64 bit variables respectively.
Like you can see in the figures.
As you can imagine, single precision with 32-bits allow you to store twice the amount of variables in the same space.
This means that you can load twice the amount of variables in the cache, and thus are more efficient when computing the data due to fewer memory loads.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

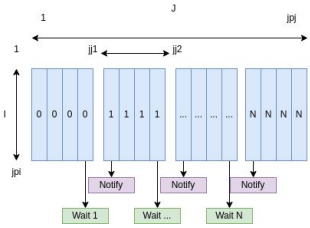
GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

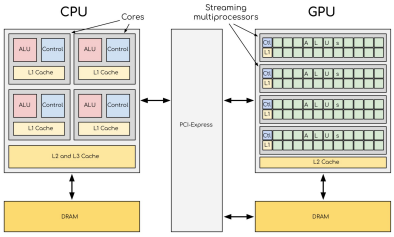
Server icon:
https://dribbble.com/shots/4189516-Parallel-Computing-Icon

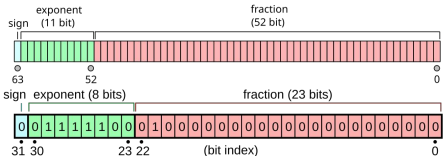
Performance Engineering

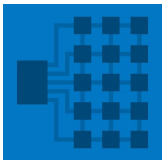
Performed optimizations:

- CPU
  - OpenMP regions (1.2x - 2.8x depending on the region)
  - MultIO improvement (1.03x)
  - Mixed Precision (1.2x)
- GPU
  - Porting pilot-regions of Sea-Ice module to GPUs (5.9x)









And lastly, my sub-team has been porting parts of the model to GPU to see if we can speedup the computation.

This resulted in a 5.9x times speedup for the first module that we ported.

GPU computing requires memory to be transferred from the CPU to the GPU, which takes time.

Hence, we hope to achieve higher global speedup when we include other modules as well that reuse the same memory.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

Server icon:
https://dribbble.com/shots/4189516-Parallel-Computing-Icon

CPU-GPU figure:
https://enccs.github.io/gpu-programming/_images/CPUAndGPU.png

Performance Profiling

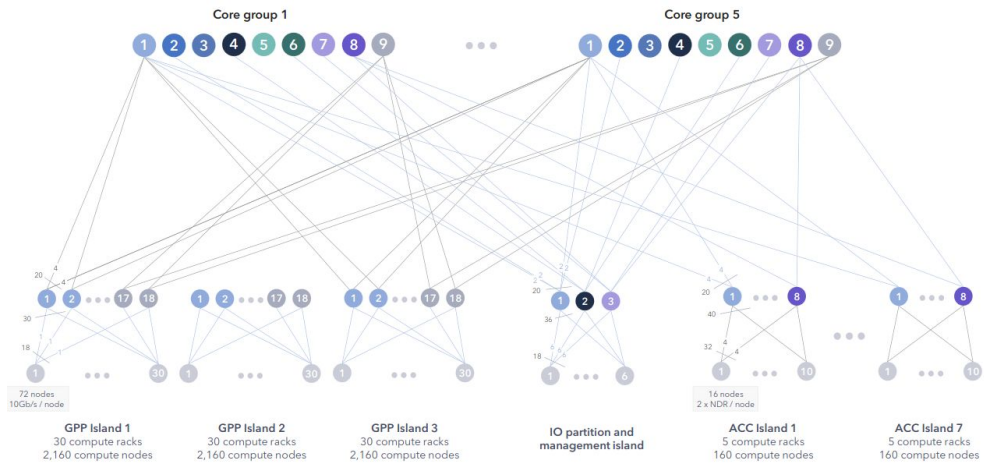
Alright, so let's take a look at how we detect inefficiencies within the models.
I'll share some of our tools and techniques we use on a daily basis.

HPC 101

However, in order to do so I'll give a super brief introduction to high performance computing,

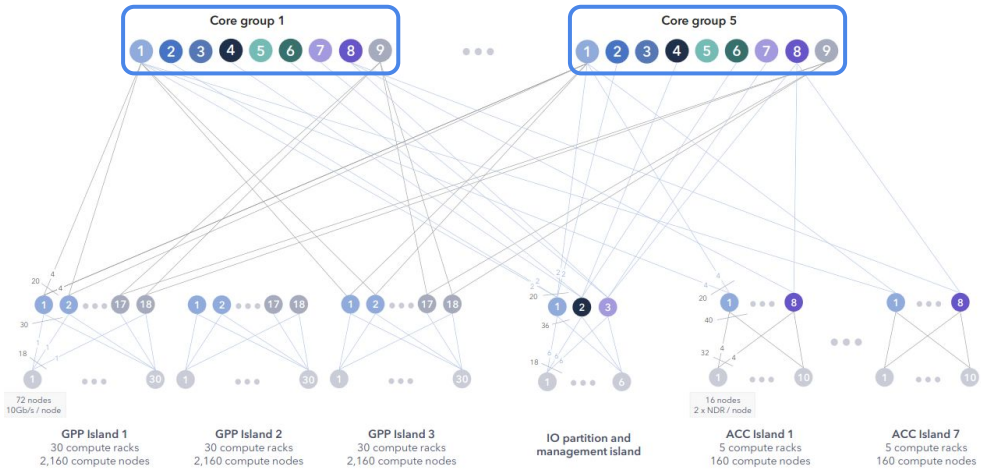
As you will need to know how supercomputers work to understand the tools.

HPC 101



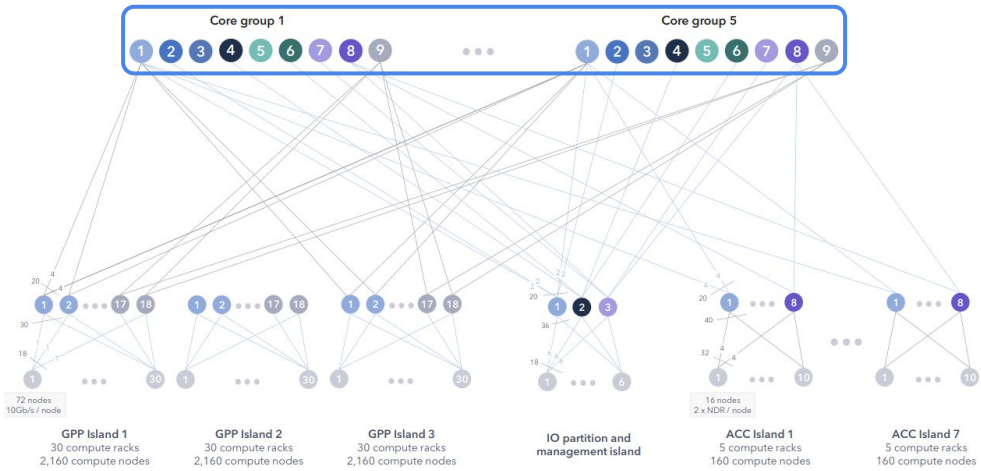
This is an overview of the MareNostrum 5 system topology.
It might look a bit daunting, but don't worry it's secretly quite simple.

HPC 101



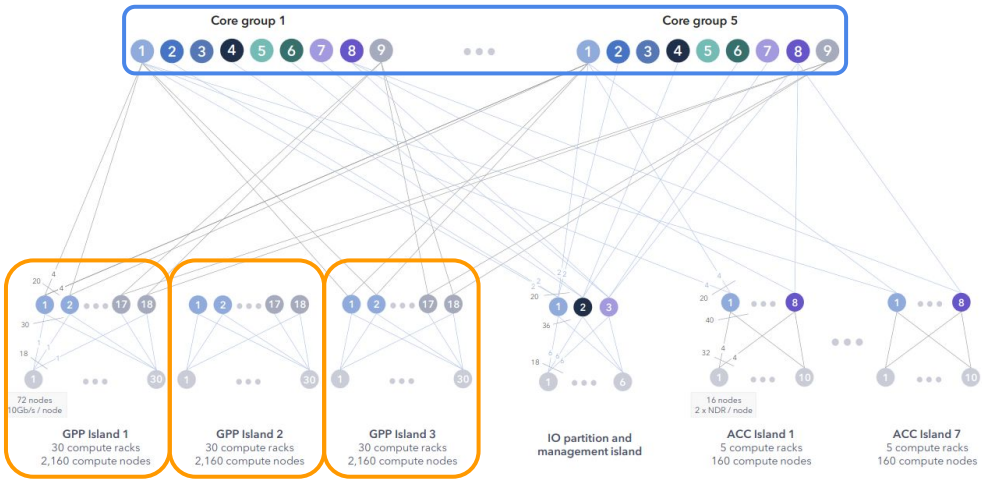
These core groups are basically groupings of different types of nodes that you can use.
A node is basically the smallest building block of a supercomputer, which consists of a CPU, Memory, and a Network card.
Some also come with GPUs.

HPC 101



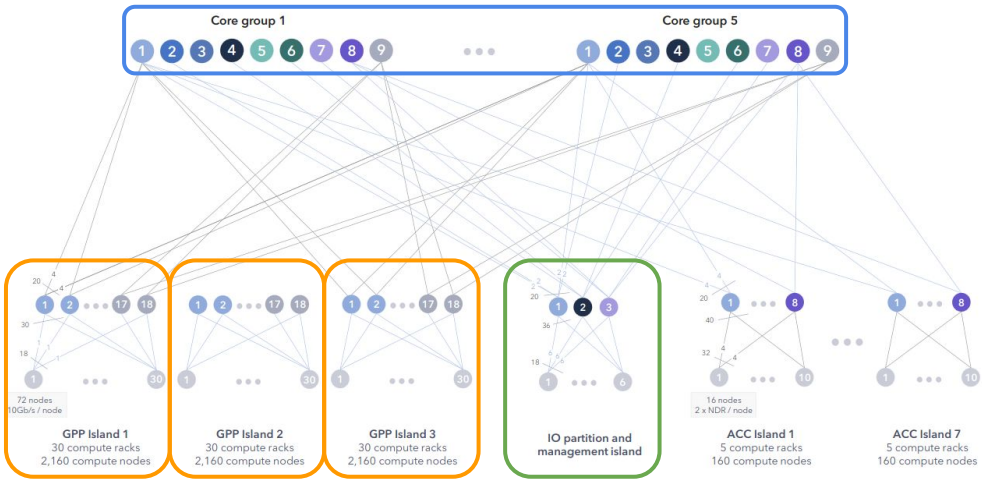
In this example we will use the core groups as the main controller of the machine which instructs the nodes that we see below.
So we say that this blue rectangle is one entity.

HPC 101



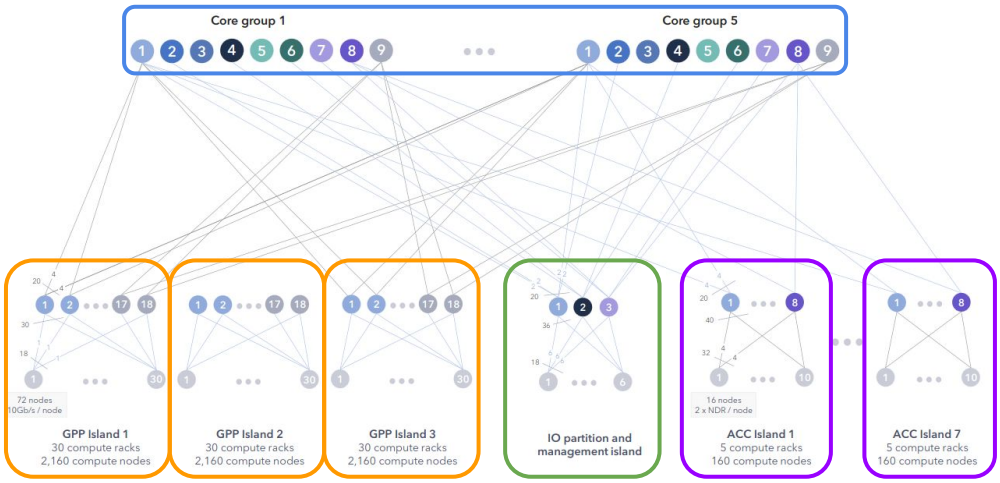
Then on the bottom left we have the GPP or so called "General Purpose Partition" islands.
These are groups of nodes that only have CPUs and no GPUs.

HPC 101



In the middle there is the IO partition, which manages reading and writing to files within storage.

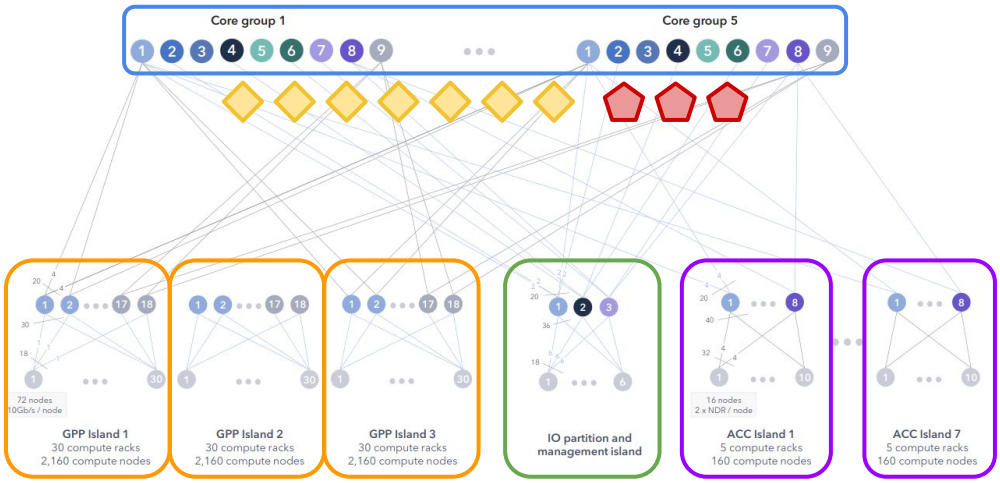
HPC 101



And on the right side we have the ACC or so called "Accelerated Partition" islands. These are groups of nodes which have both CPUs and GPUs available. Thence, when using the supercomputer, it is these nodes that you use for GPU computing.

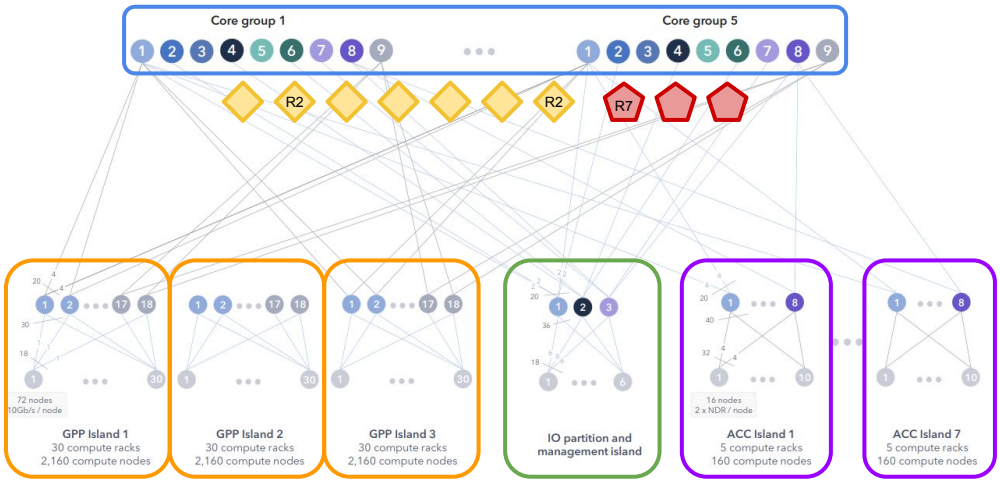
For each of these islands, we simply say that each rectangle is one nodfe, just as we did for the core groups.

HPC 101



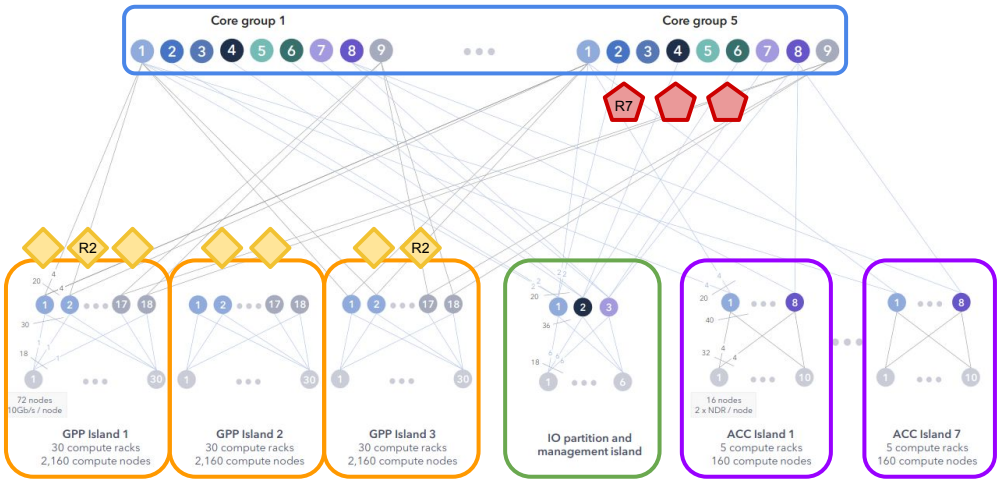
Let's add some work to the picture.
The yellow diamonds are CPU work, and the red hexagons are GPU work.
So these are parts of a problem that we want to compute.

HPC 101



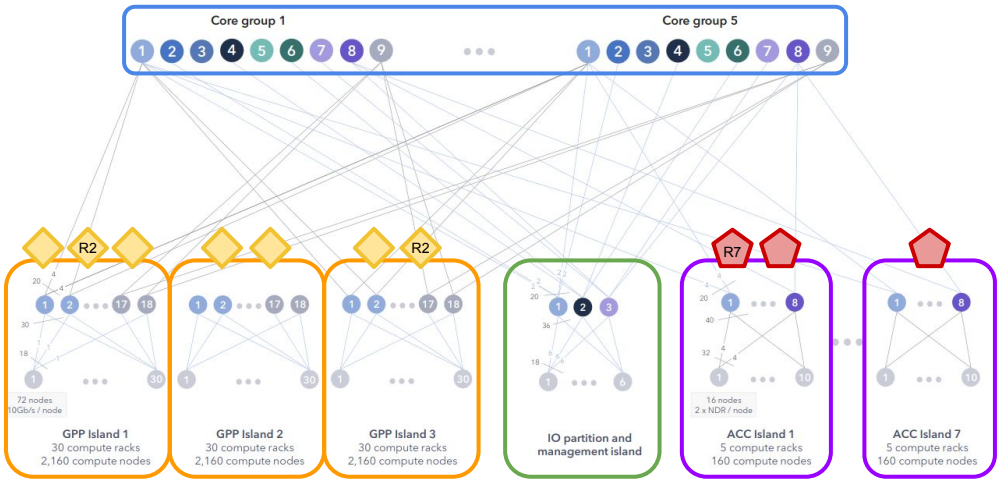
Sometimes, a piece of work needs the results from another piece of work before it can start.
I've written that as R followed with a number, where the number represents the island it needs the data from.

HPC 101

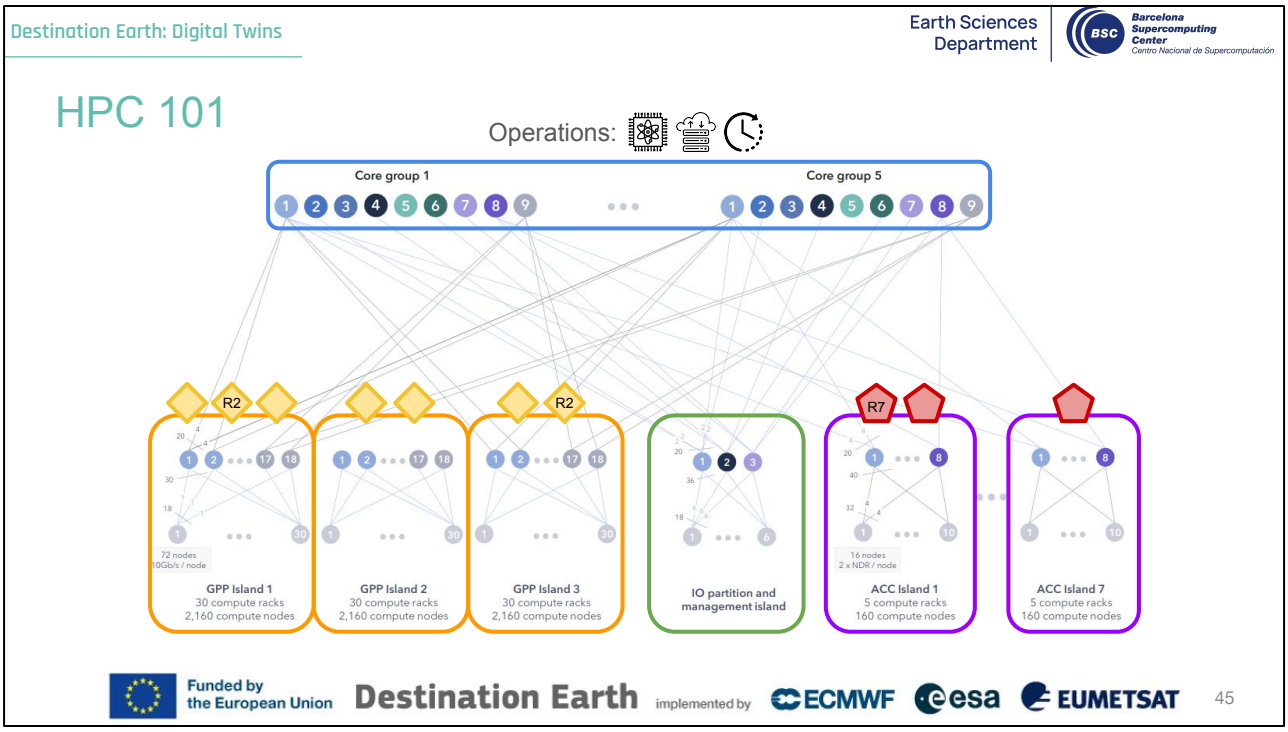


Let's distribute the CPU work among our nodes.

HPC 101



And do the same for our GPU work.



Within a supercomputer, nodes can do a couple different operations.
For this simplified example, we limit us to:

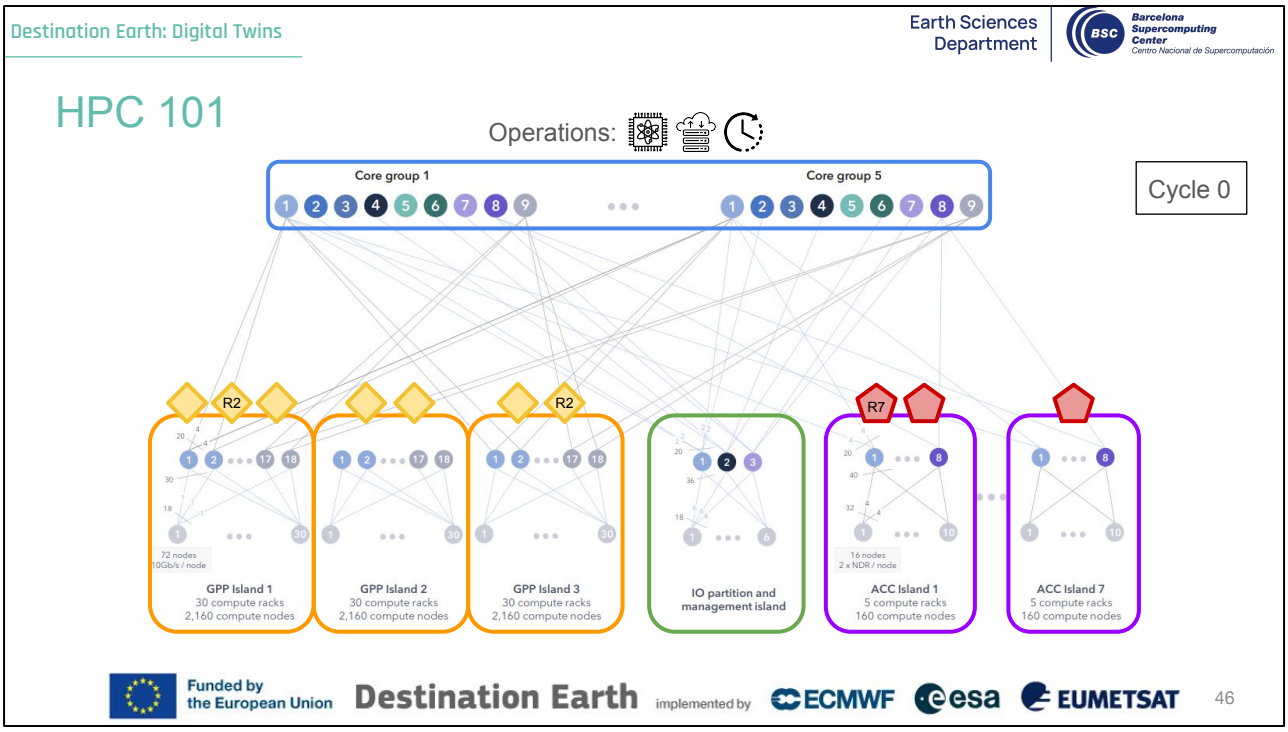
Computing work

Transferring data

And waiting

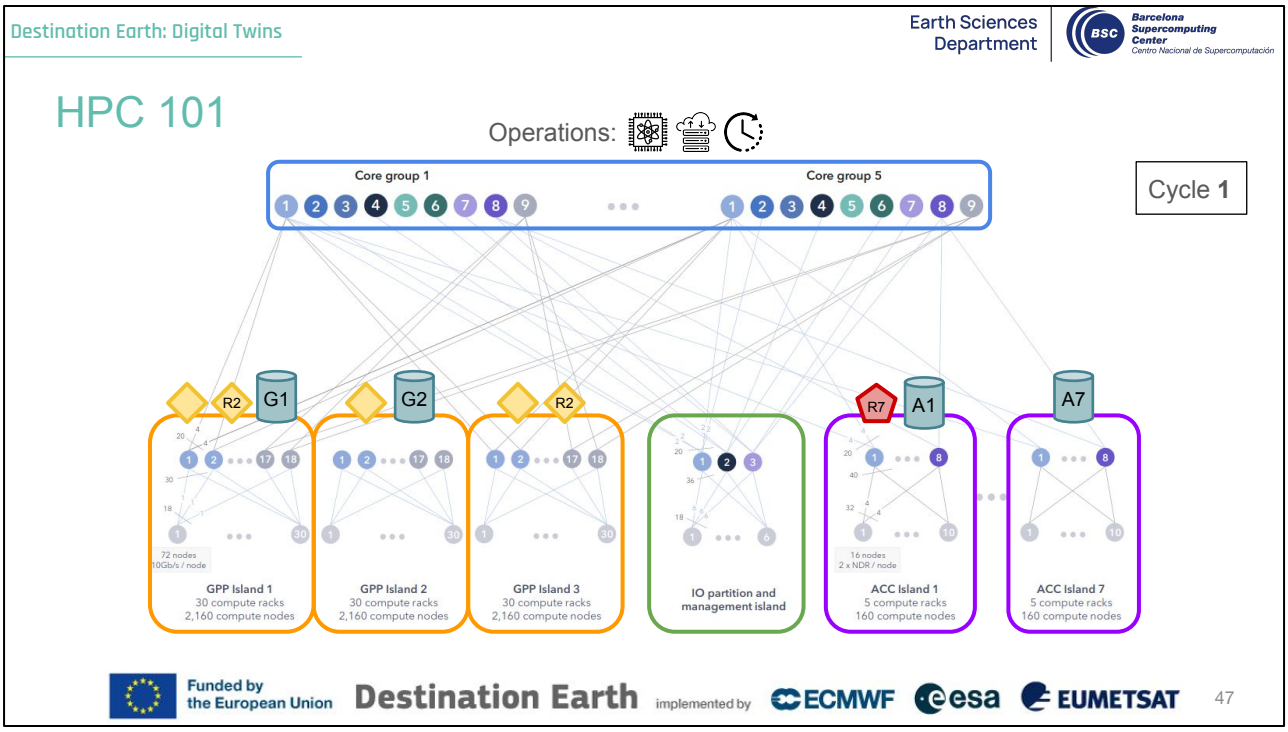
Icons:

- Time left: https://www.flaticon.com/free-icon/time-left_66163
- Microprocessor: https://www.flaticon.com/free-icon/microprocessor_5905354
- Cloud: https://www.flaticon.com/free-icon/cloud_860276



To help you understand where inefficiencies within supercomputers come from, we'll play out this little example.
On the right top side we keep track of what cycle we are currently working in.
A node can only do one thing in a cycle, which is compute work, transfer data, or wait.

So we get into cycle one and the nodes get busy.
Watch what happens when we go into cycle 1.

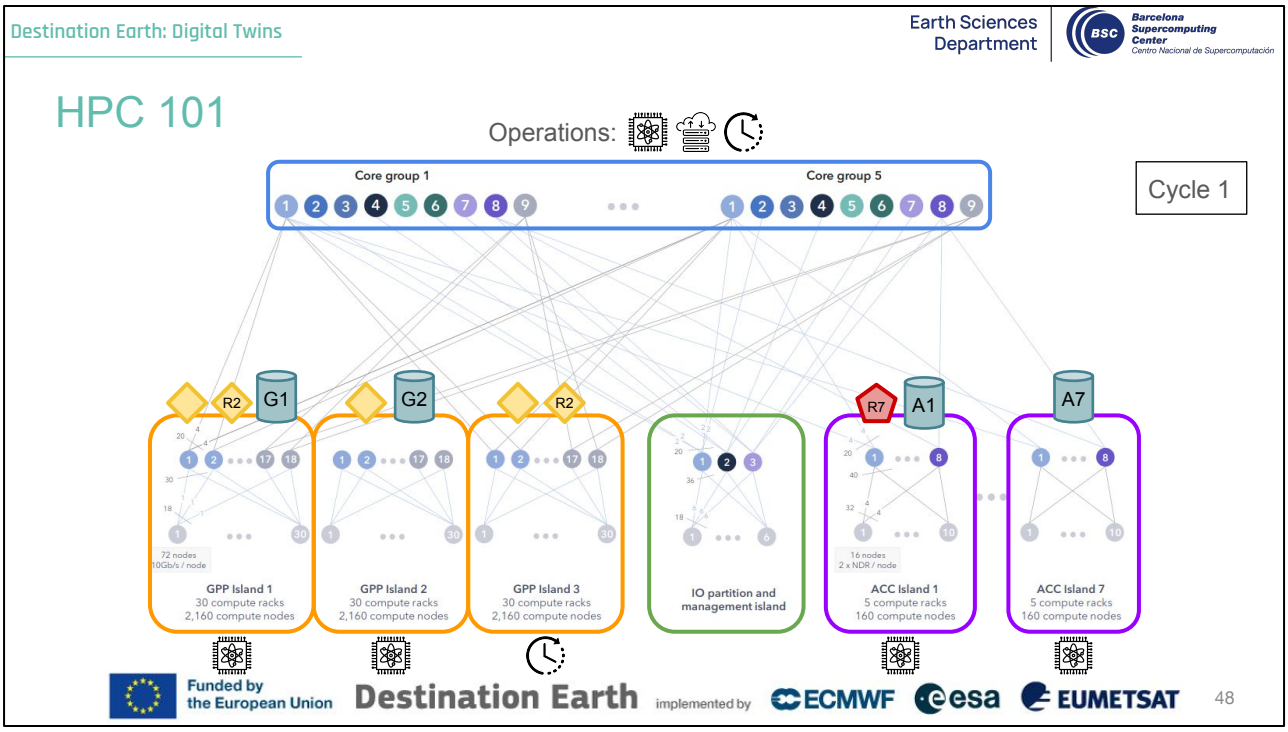


As you can see, some of the icons changed from shape into a cylinder.
We'll use this to represent solutions to the processed work.
Within the cylinder we write where the work was performed,

So for the GPP partitions we have a G,

And for the ACC partitions we have an A.

So what operations did we perform?



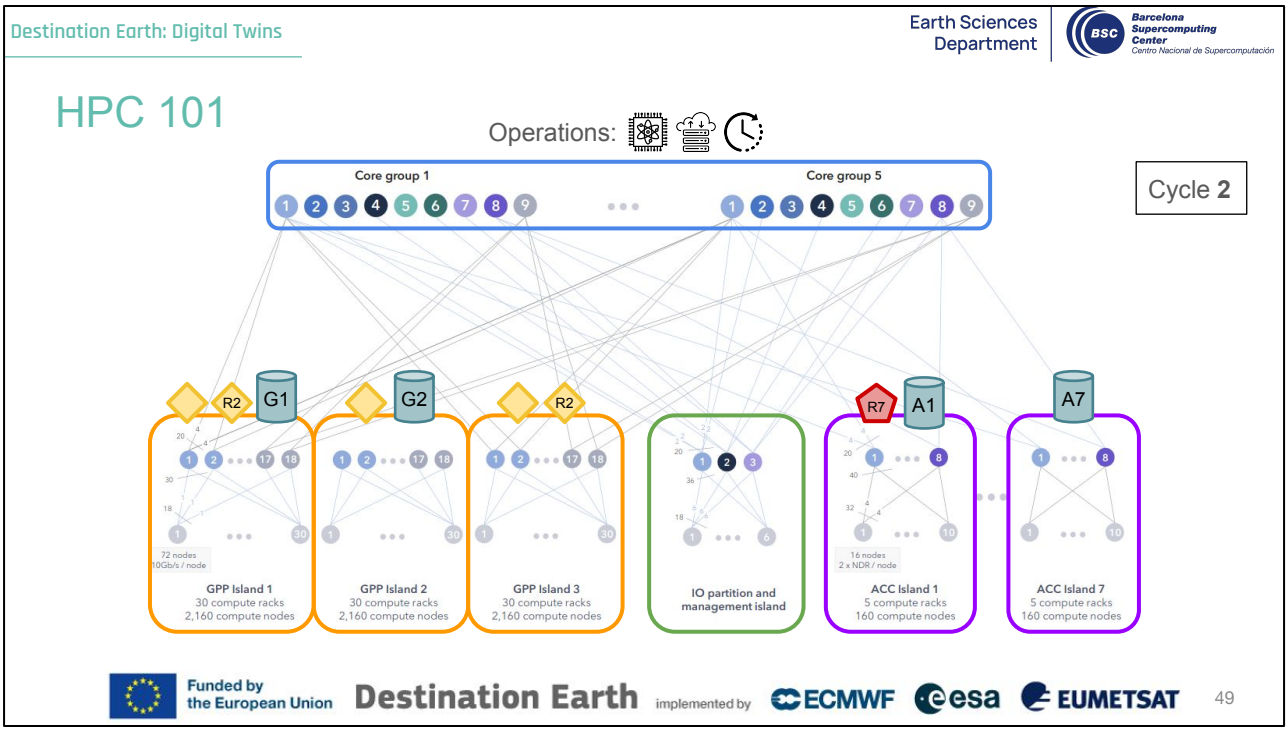
Well, the GPP 1 computed their problem into a solution.

The same applies for GPP 2.

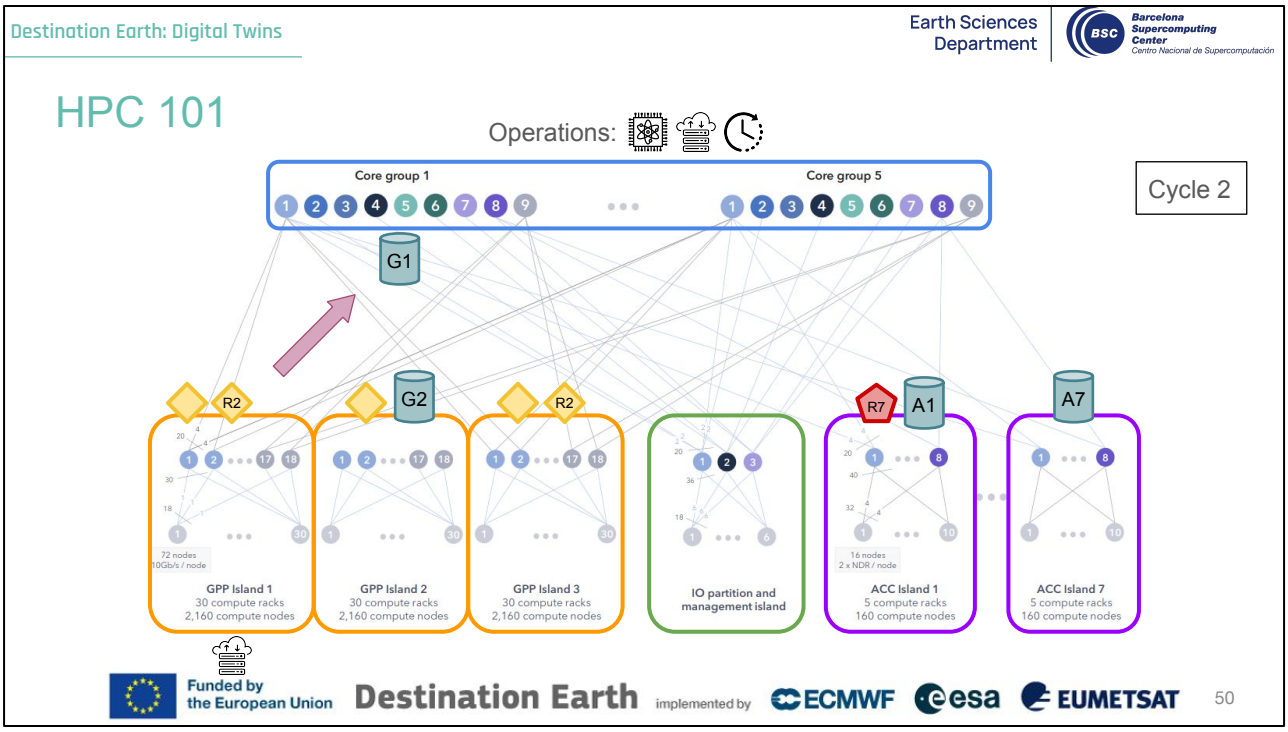
For GPP 3, nothing happened.
It had to wait. Why?
Well, because it needs the results from GPP 2 before it can start computing.
So you can already see that waiting for each other is inefficient.

Then on the right side we have ACC 1 which could compute.

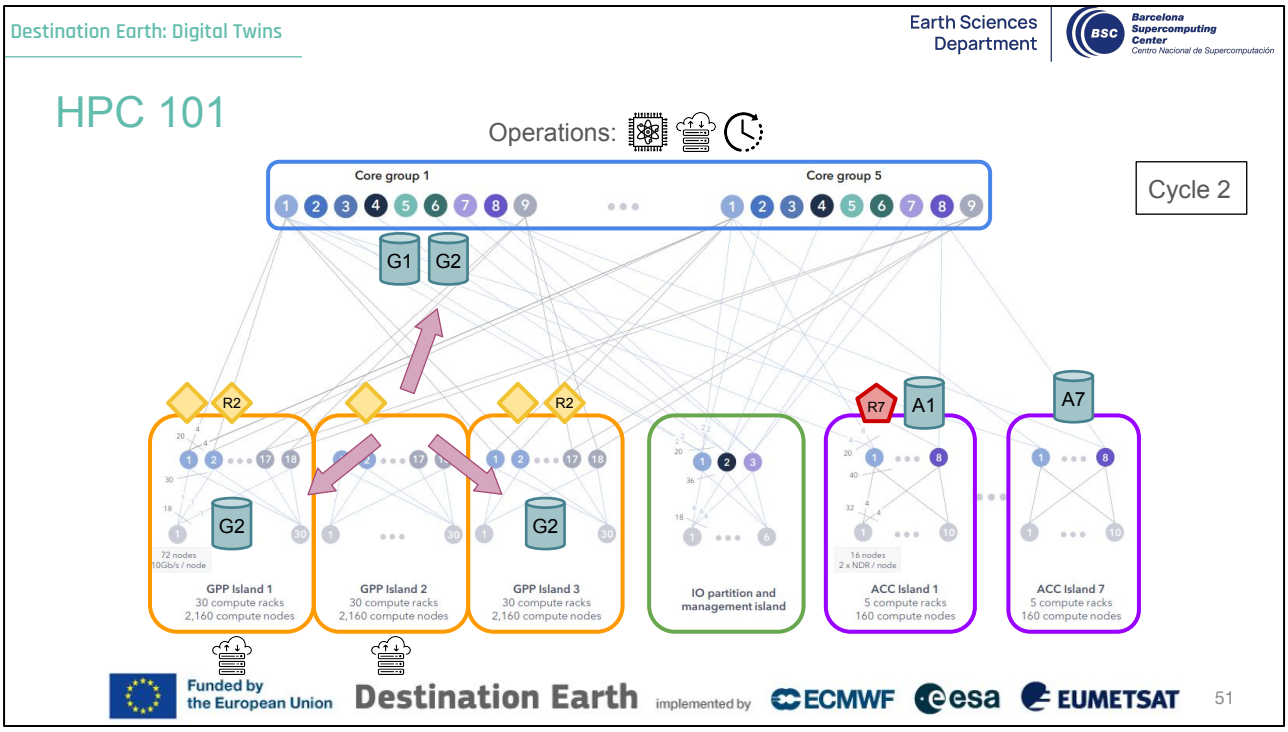
And the same is the case for ACC 2.



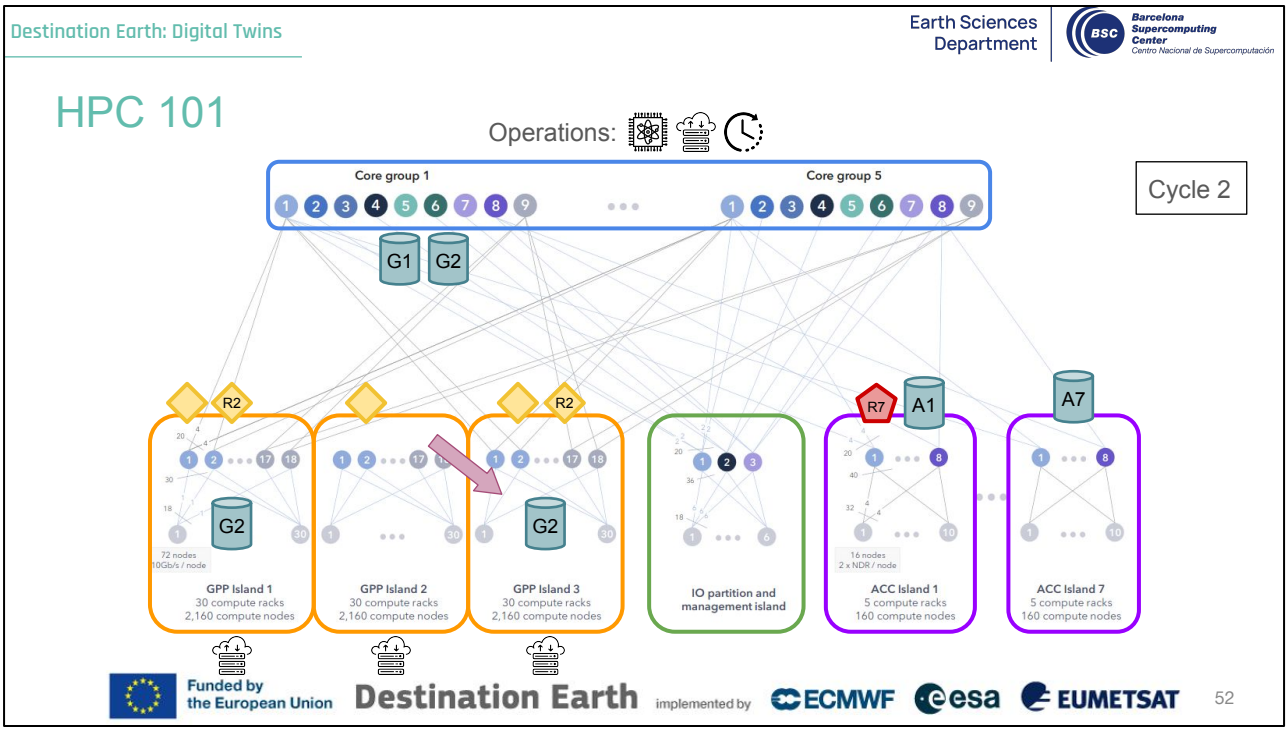
Then we go to cycle 2.



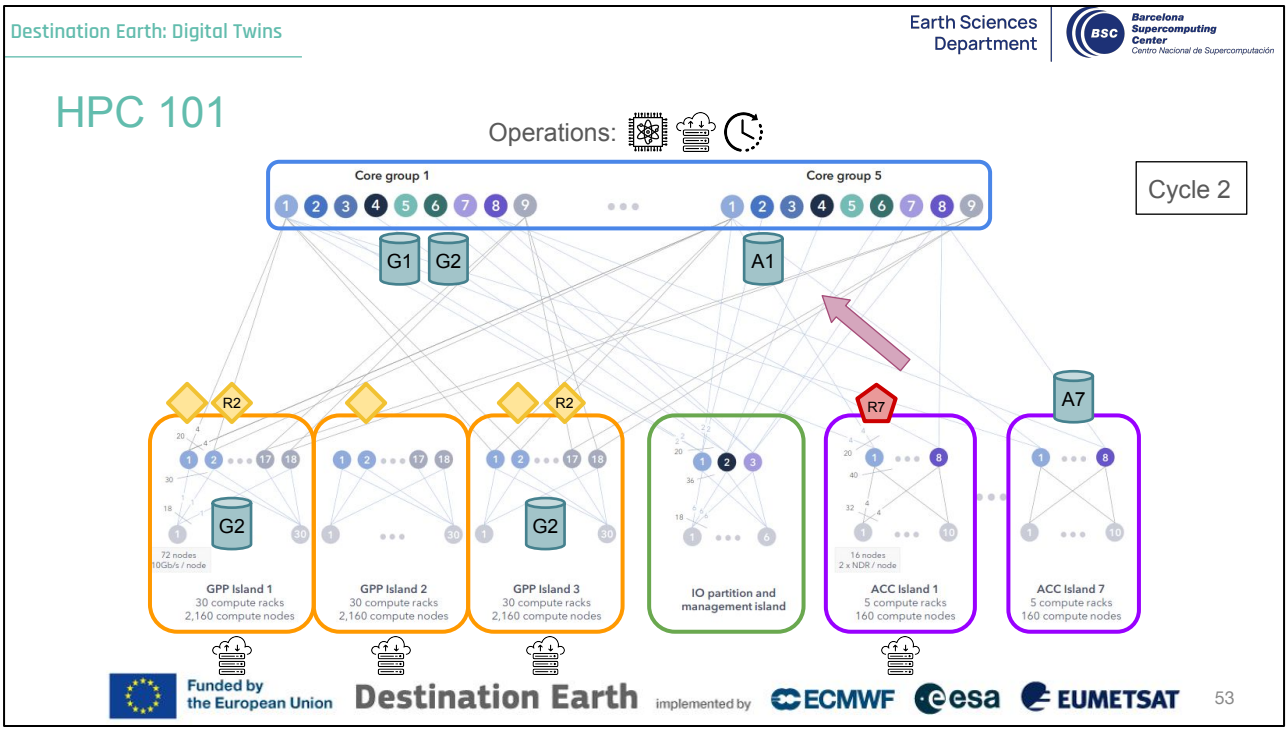
GPP 1 node has data that can be returned to the controller.
Thus it's performing a data transfer operation.



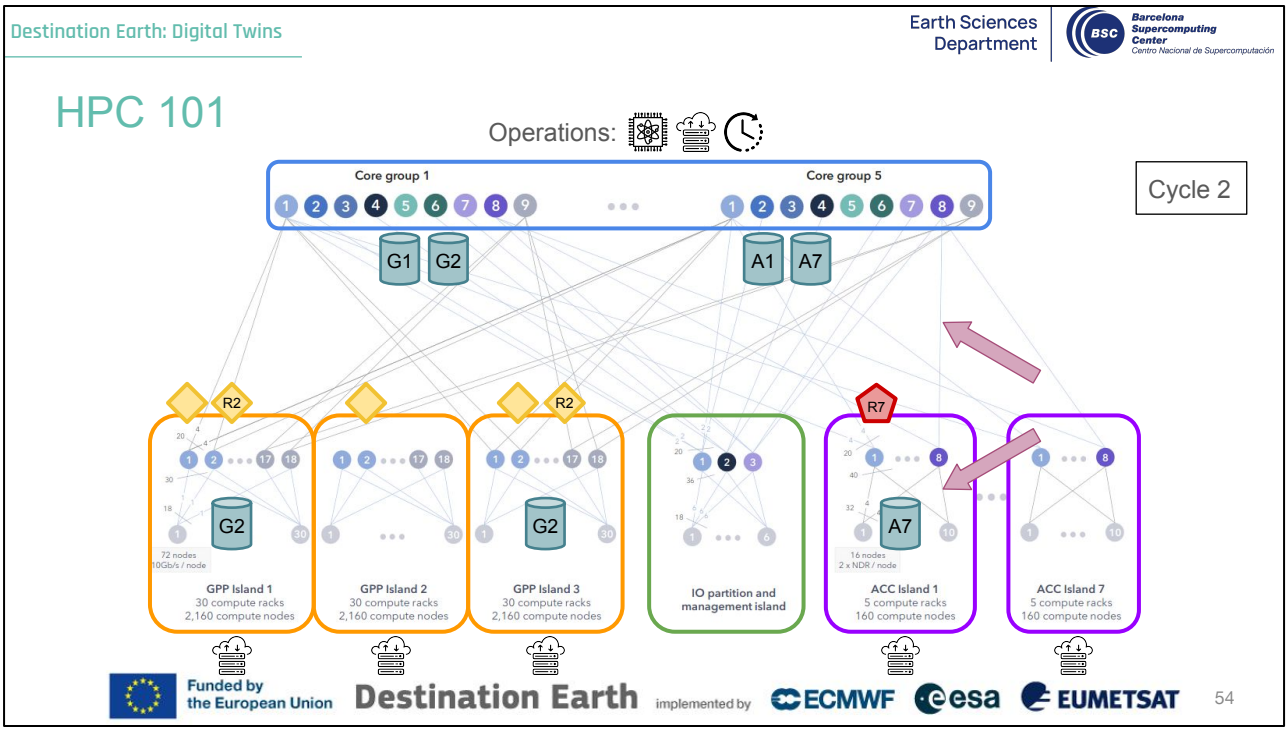
The same applies for GPP 2.
However, now we see that GPP 1 and GPP 3 need input data from GPP 2.
Thus, we not only transfer the results back, but we also send them to the other nodes.



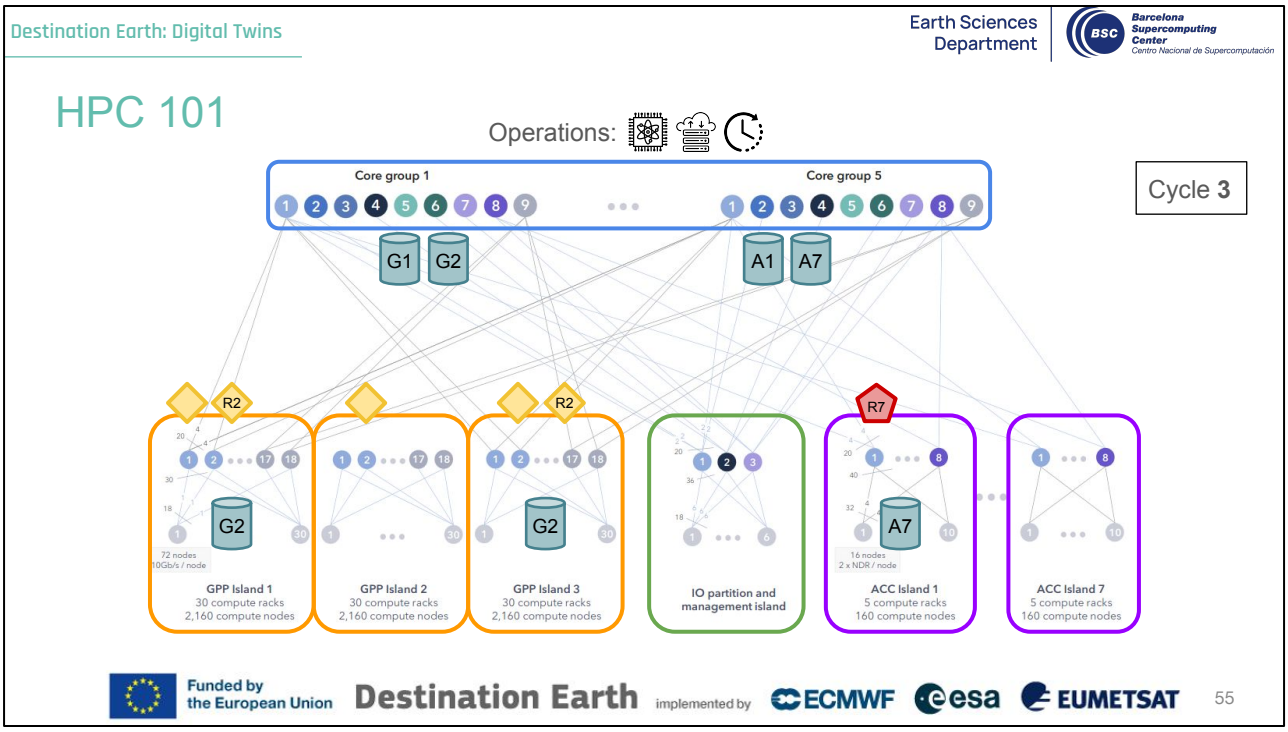
Then for GPP 3, you might think it waits, but now it spends its time on data transfer as well.
But then the transfer is receiving data.



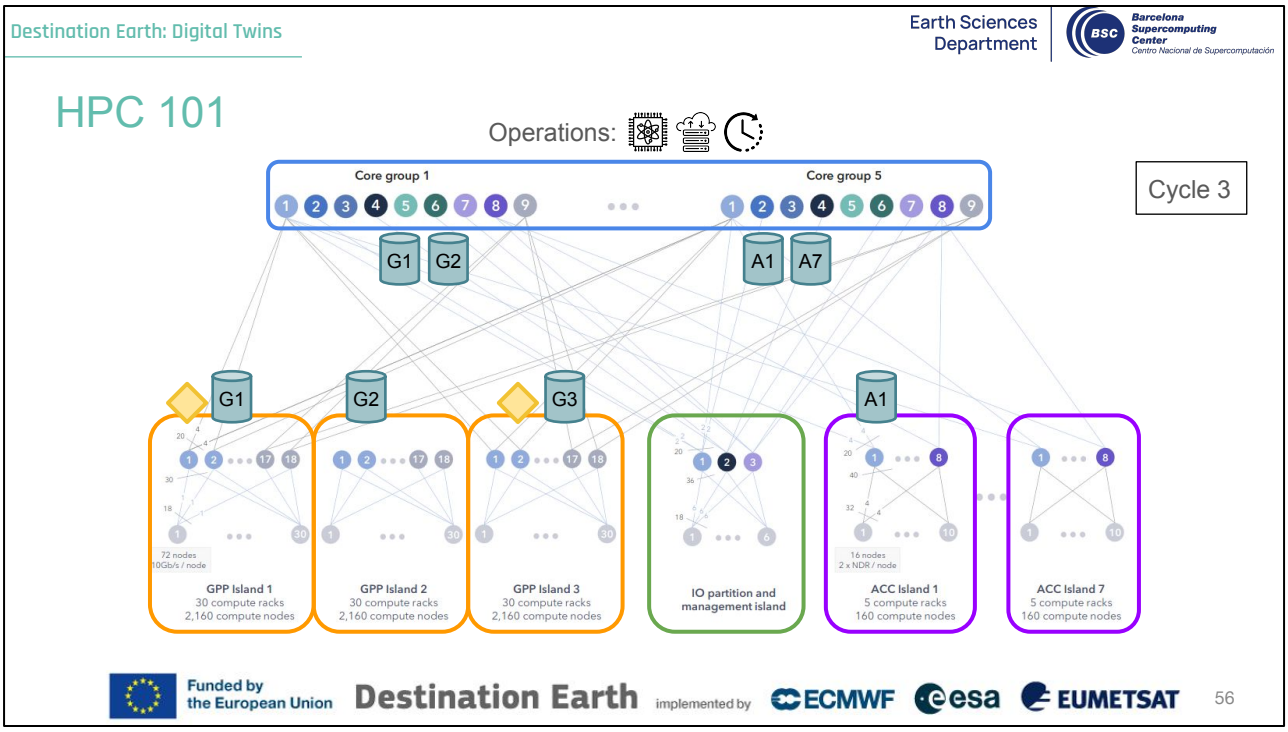
For ACC 1 we have results, which can also be send back to the controller.



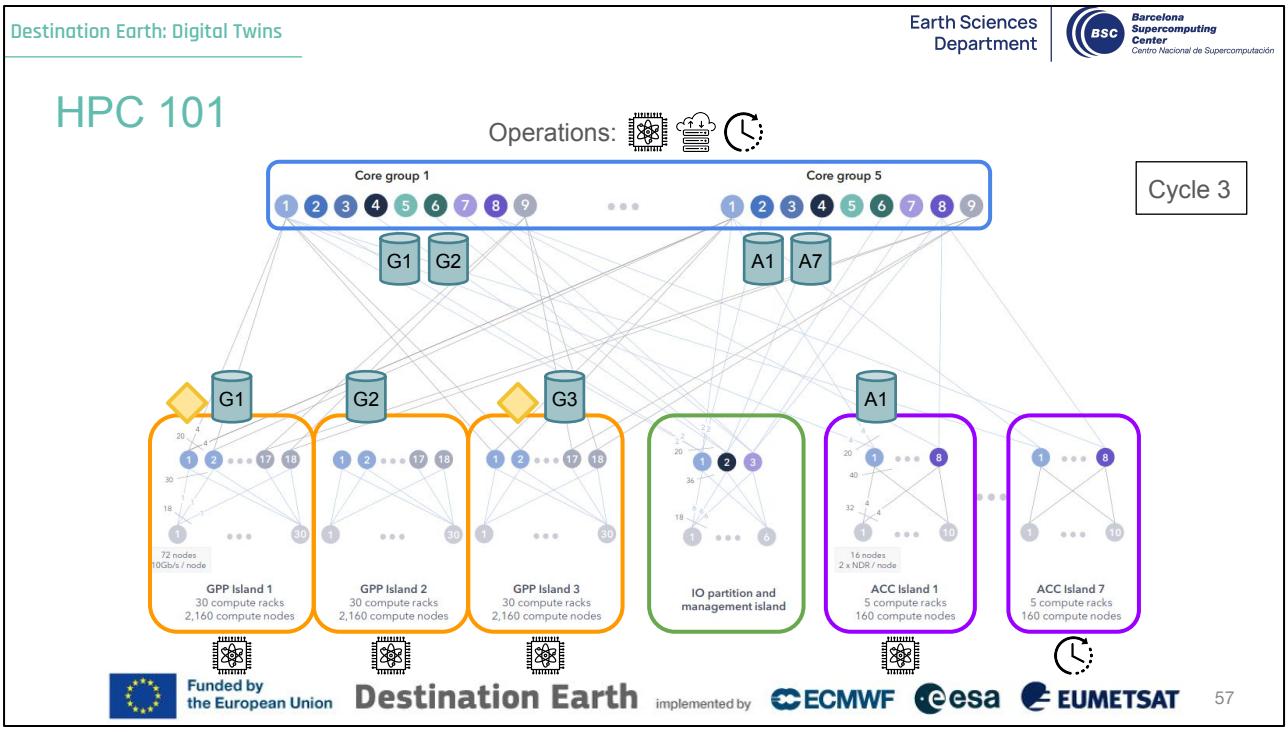
And ACC 7 has results that ACC 1 needs, thus we see again a data transfer that goes to both the controller and another node.



And we continue to cycle 3.

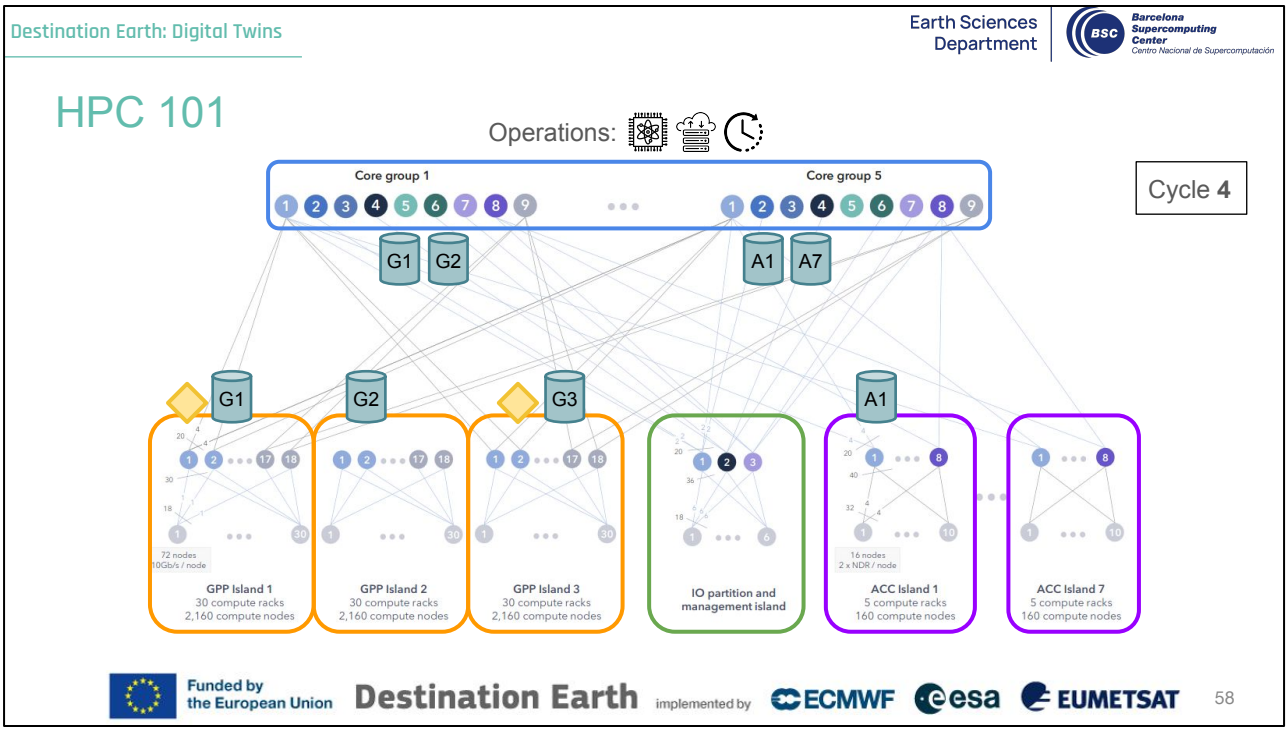


We again perform the operations, nothing needed to be transferred.

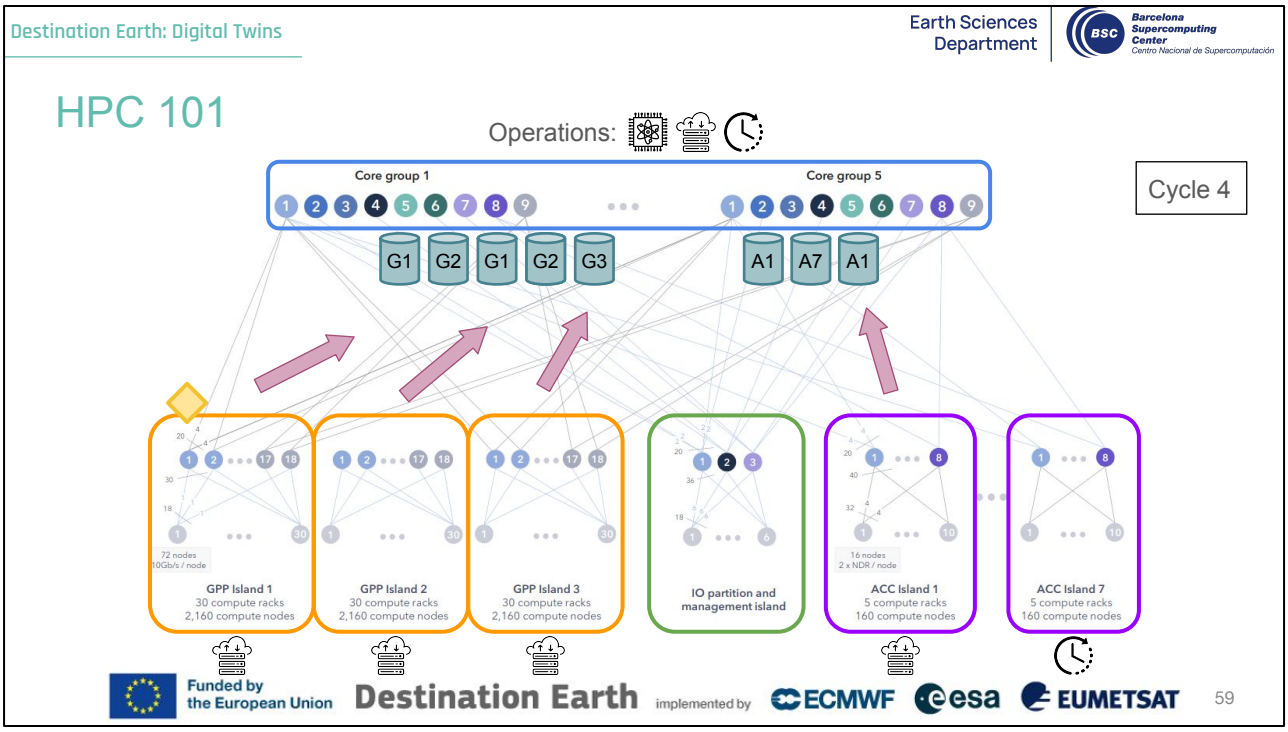


And thus we see, again that all nodes could spend time computing, except for ACC 7. Why? Well, because ACC 7 has no work anymore.

So this is another inefficiency that we have to watch out for.
We could this the load balancing, where we have an imbalance if some nodes have more work to do than other nodes.

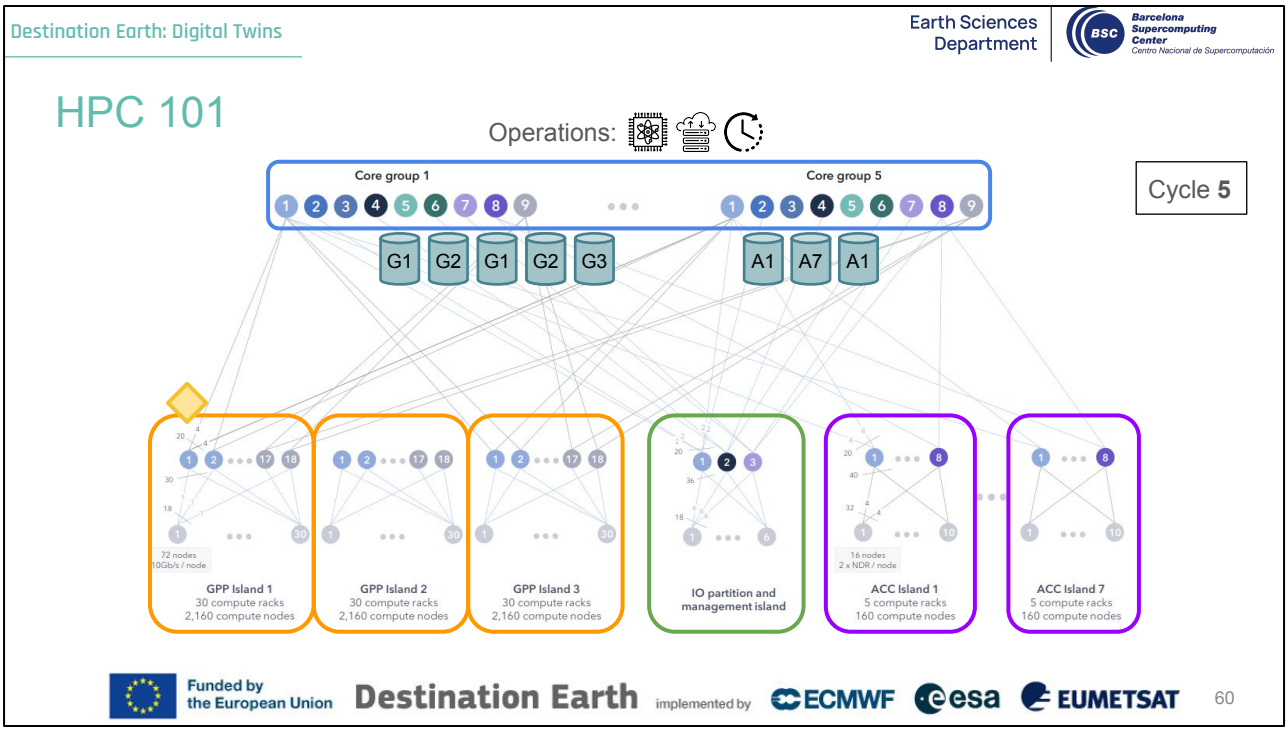


Going to cycle 4.

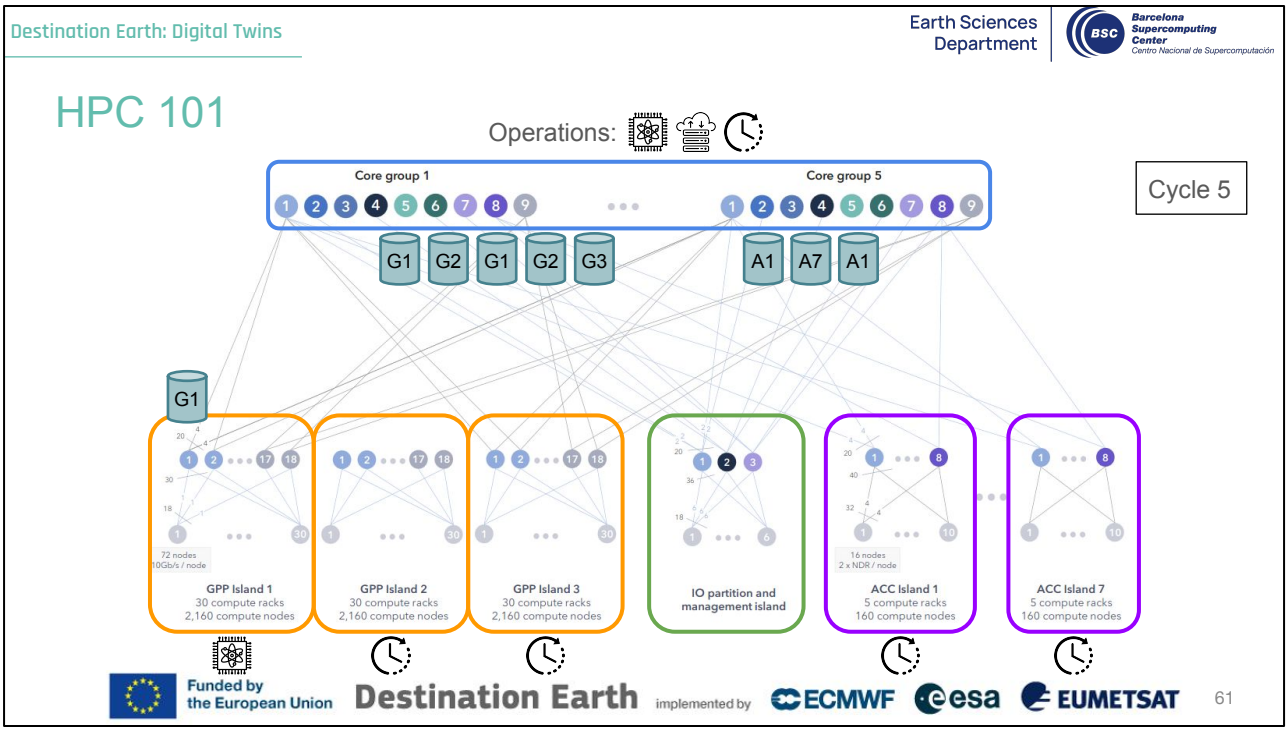


All of the results that we computed in cycle 3 need to be transferred to the controller. There is no node that requests any input from another node.

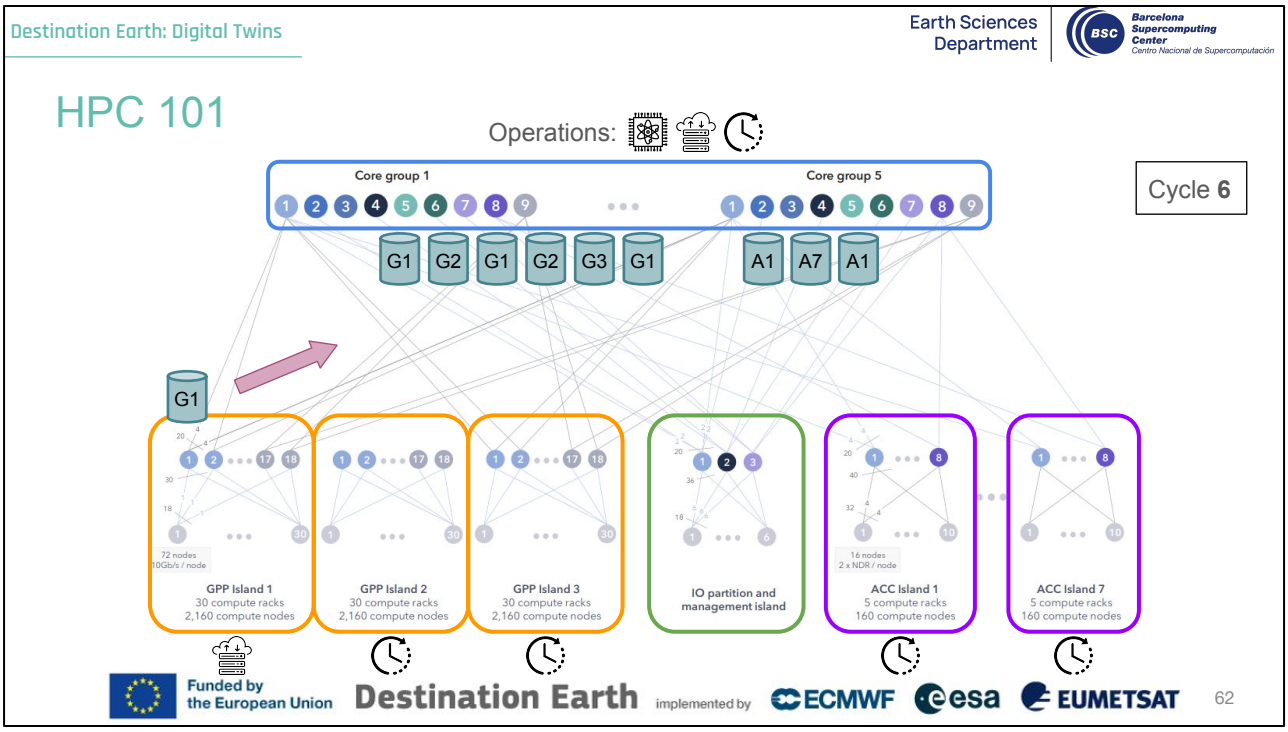
ACC 7 again waits.



Then for cycle 5, everything looks quite empty.



GPP 1 computes the last piece of work it has, but all the other nodes spend their cycle waiting.
And now you can see that the load imbalance can have quite a big impact on the system.

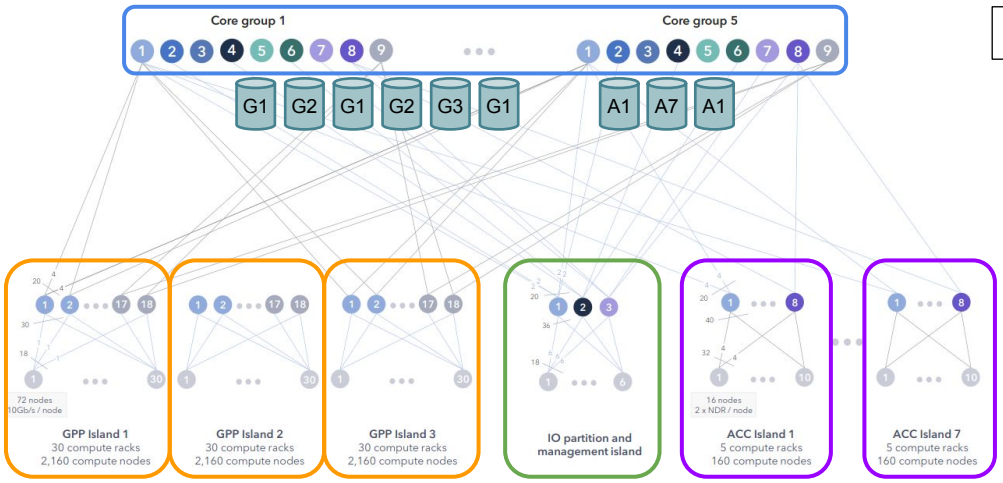


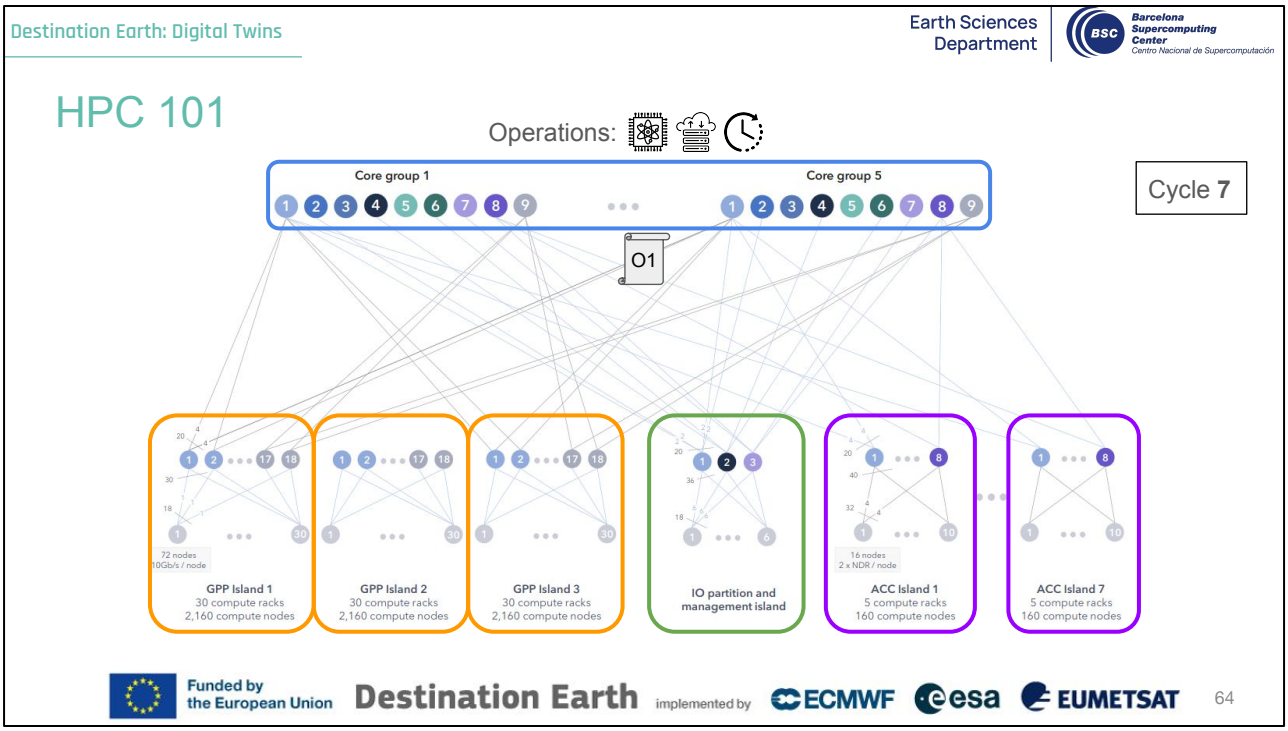
In cycle 6 we move the result of GPP 1 to the controller.

HPC 101

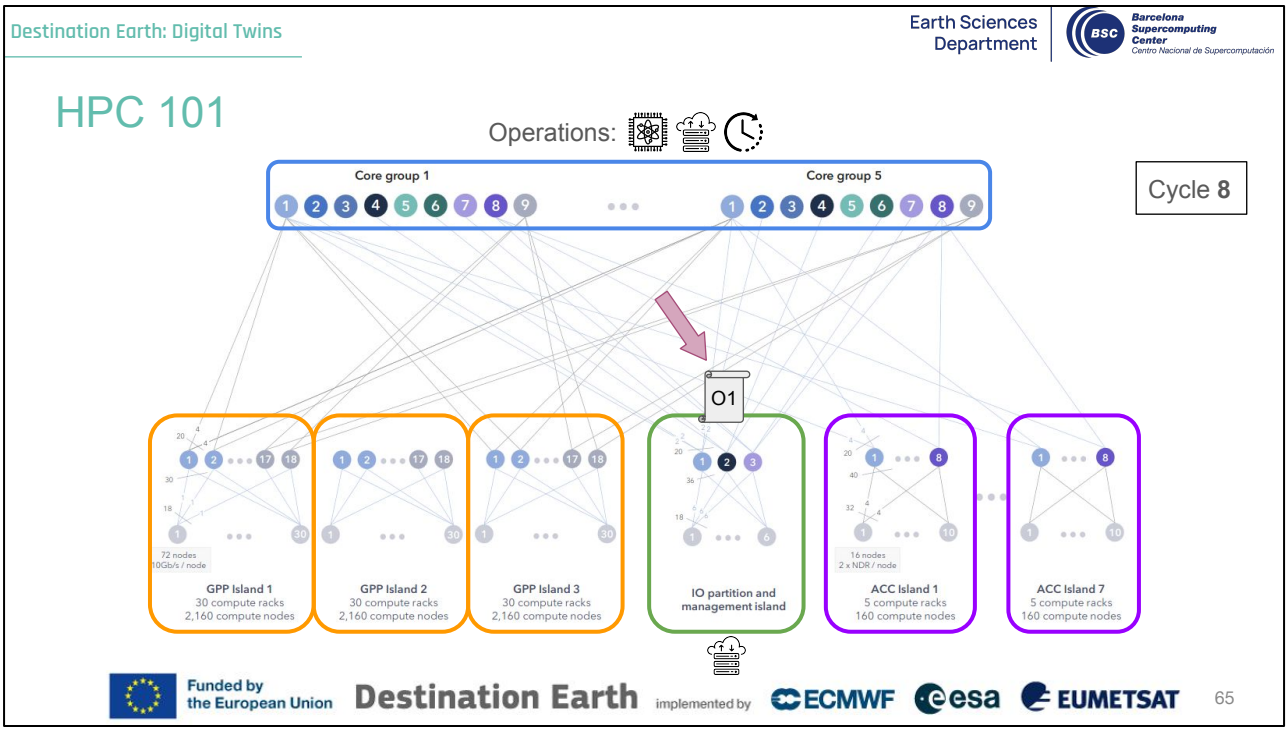
Operations:

Cycle 6

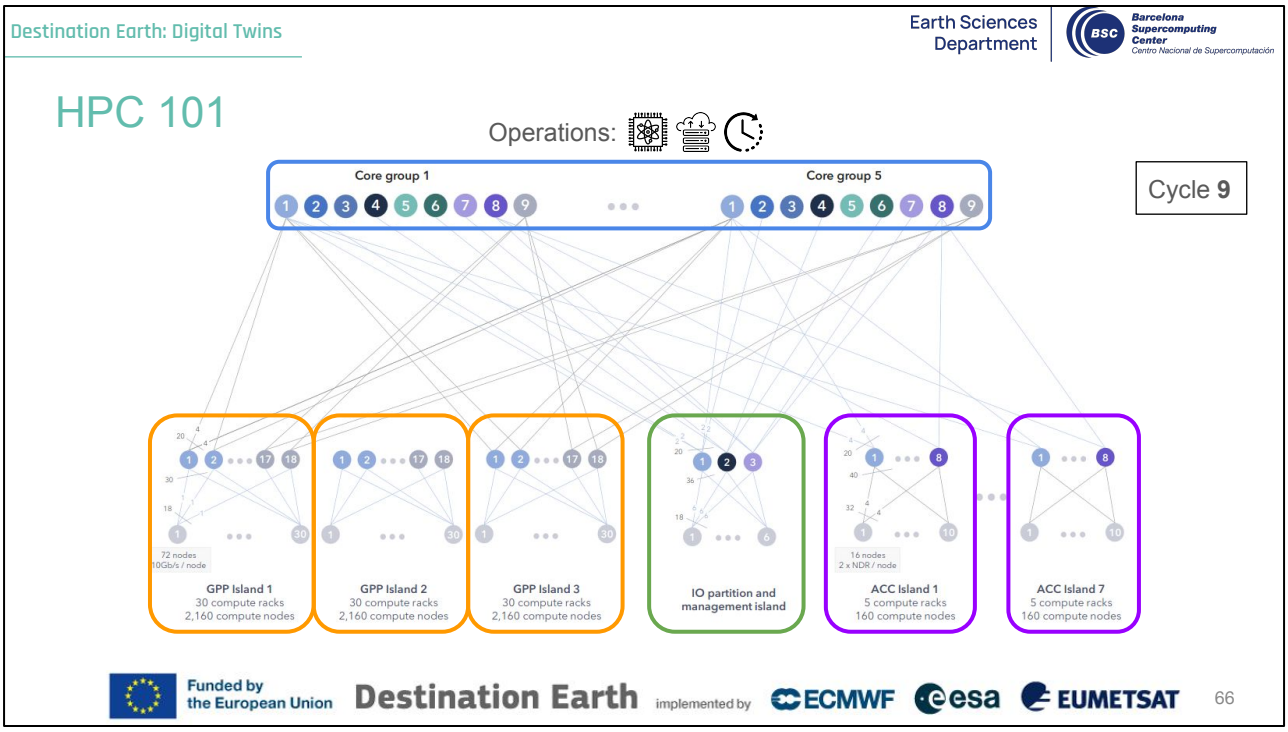




And then in cycle 7, we convert all the results in output that can be written to a file.



This is moved to the IO partition in cycle 8.



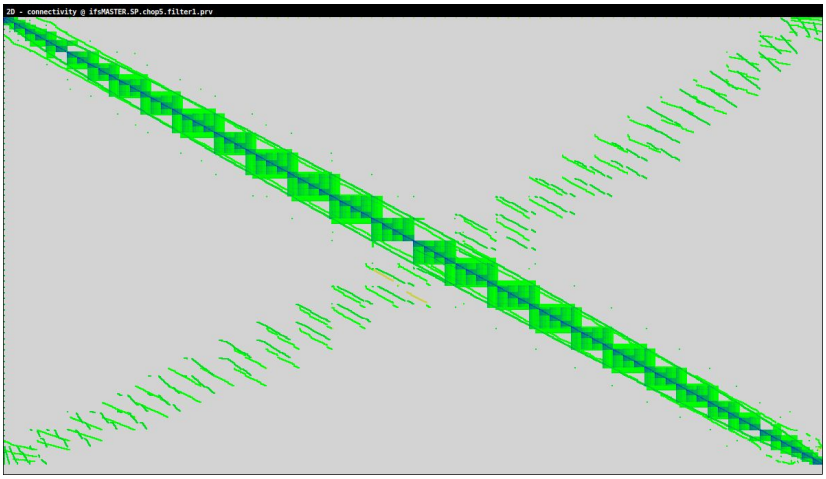
And in cycle 9 we write the file to disk, finishing the execution.

As you can see, there is a lot of communication going on with the sending and receiving of data.

Performance Profiling

Now that you sort of know how a high-performance computer works, we can talk about Performance Profiling.
This is basically the task of analyzing the model in order to find potential inefficiencies.
Most of the times, we visualize the data we collect in order to make it easier.
Hence I will show you many visualizations and explain what you are looking at.
As a first example, the communication pattern that you saw on the previous slide is one of those visualizations.

Performance profiling



So in order to perform performance profiling, we need some tools.

Performance profiling

Tools to use:

So in order to perform performance profiling, we need some tools.

Performance profiling

Tools to use:

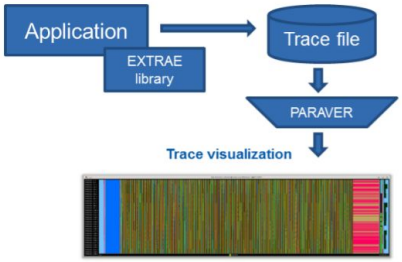
- GPU → Depends on platform!

For the GPU, it really matters what GPU vendor you have.
NVIDIA has their Nsight platform, while AMD has their ROCm platform.

Performance profiling

Tools to use:

- GPU → Depends on platform!
- CPU → Extrae & Paraver



For CPU, you usually can use a tool for multiple platforms, like Intel and AMD.
We have two in-house open-source tools, called Extrae and Paraver.
Extrae is used to collect information on the model.
And Paraver is used to visualize that data.

In this visualization you can see how Extrae acts as a library that is loaded during the application execution.
This produces a so-called trace file of the execution.
Then we feed the trace file to Paraver, which visualizes it in a timeline.
Later you'll see more examples of these trace files, so don't worry if it sounds too vague.

CPU Profiling

- CPU Modelfactors

	10240(1280x8)[1]	20480(2560x8)[2]
Global efficiency	60.99	43.63
-- Parallel efficiency	60.99	48.09
-- Load balance	84.62	84.07
-- Communication efficiency	72.07	57.20
-- Computation scalability	100.00	90.74
-- IPC scalability	100.00	90.46
-- Instruction scalability	100.00	98.44
-- Frequency scalability	100.00	101.89

	10240(1280x8)[1]	20480(2560x8)[2]
-- Hybrid Parallel efficiency	60.99	48.09
-- MPI Parallel efficiency	68.40	55.96
-- MPI Load balance	86.62	85.99
-- MPI Communication efficiency	78.97	65.08
-- Serialization efficiency		
-- Transfer efficiency		
-- OpenMP Parallel efficiency	89.16	85.93
-- OpenMP Load Balance	97.69	97.77
-- OpenMP Communication efficiency	91.27	87.90

Percentage(%): 100, 80, 60, 40, 20, 0

The first visualization I want to show you, is what we call "Modelfactors". Normally it is 1 big table, but due to the slides I chopped it up in 2 parts.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- CPU Modelfactors

	10240(1280x8)[1]	20480(2560x8)[2]
Global efficiency	60.99	43.63
-- Parallel efficiency	60.99	48.09
-- Load balance	84.62	84.07
-- Communication efficiency	72.07	57.20
-- Computation scalability	100.00	90.74
-- IPC scalability	100.00	90.46
-- Instruction scalability	100.00	98.44
-- Frequency scalability	100.00	101.89

Percentage(%): 100, 80, 60, 40, 20, 0

	10240(1280x8)[1]	20480(2560x8)[2]
-- Hybrid Parallel efficiency	60.99	48.09
-- MPI Parallel efficiency	68.40	55.96
-- MPI Load balance	86.62	85.99
-- MPI Communication efficiency	78.97	65.08
-- Serialization efficiency		
-- Transfer efficiency		
-- OpenMP Parallel efficiency	89.16	85.93
-- OpenMP Load Balance	97.69	97.77
-- OpenMP Communication efficiency	91.27	87.90

Percentage(%): 100, 80, 60, 40, 20, 0

The Modelfactors compute the efficiency of your code for different workloads.
So in these rows you can see the global efficiency for two system setups, namely 10240 and 20480 cores.

Basically 80 and 160 nodes on HPC2020, 128 cores per node.

These numbers correspond to the number of nodes that we have used, so the 2nd column distributes the work between more parts of the supercomputer than the first one.

You can see that the global efficiency drops from 60.99% to 43.63%.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- CPU Modelfactors

	10240(1280x8)[1]	20480(2560x8)[2]
Global efficiency	60.99	43.63
-- Parallel efficiency	60.99	48.09
-- Load balance	84.62	84.07
-- Communication efficiency	72.07	57.20
-- Computation scalability	100.00	90.74
-- IPC scalability	100.00	90.46
-- Instruction scalability	100.00	98.44
-- Frequency scalability	100.00	101.89

Percentage(%): 100, 80, 60, 40, 20, 0

	10240(1280x8)[1]	20480(2560x8)[2]
-- Hybrid Parallel efficiency	60.99	48.09
-- MPI Parallel efficiency	68.40	55.96
-- MPI Load balance	86.62	85.99
-- MPI Communication efficiency	78.97	65.08
-- Serialization efficiency		
-- Transfer efficiency		
-- OpenMP Parallel efficiency	89.16	85.93
-- OpenMP Load Balance	97.69	97.77
-- OpenMP Communication efficiency	91.27	87.90

Percentage(%): 100, 80, 60, 40, 20, 0

The rows underneath are sub-sections, which each describe a different aspect of the efficiency.
Together they form the percentage of the row above,
so the 60.99% of Parallel Efficiency, with the 100% of Computation Scalability, with the 60.99% Hybrid Parallel Efficiency form the Global Efficiency.
And basically, you can go through each row to figure out where the inefficiency is coming from.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- CPU Modelfactors

	10240(1280x8)[1]	20480(2560x8)[2]
Global efficiency	60.99	43.63
-- Parallel efficiency	60.99	48.09
-- Load balance	84.62	84.07
-- Communication efficiency	72.07	57.20
-- Computation scalability	100.00	90.74
-- IPC scalability	100.00	90.46
-- Instruction scalability	100.00	98.44
-- Frequency scalability	100.00	101.89

Percentage(%): 100, 80, 60, 40, 20, 0

	10240(1280x8)[1]	20480(2560x8)[2]
-- Hybrid Parallel efficiency	60.99	48.09
-- MPI Parallel efficiency	68.40	55.96
-- MPI Load balance	86.62	85.99
-- MPI Communication efficiency	78.97	65.08
-- Serialization efficiency		
-- Transfer efficiency		
-- OpenMP Parallel efficiency	89.16	85.93
-- OpenMP Load Balance	97.69	97.77
-- OpenMP Communication efficiency	91.27	87.90

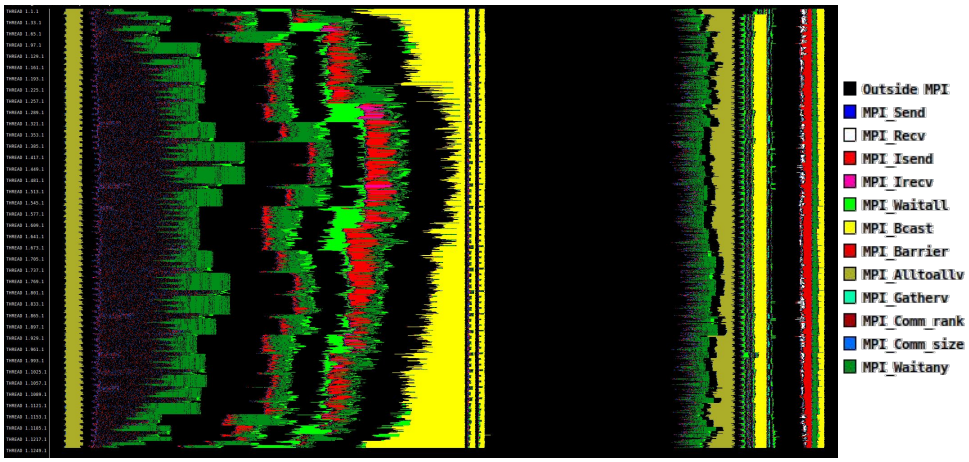
Percentage(%): 100, 80, 60, 40, 20, 0

In this case, we can see that the inefficiency mainly comes from a decrease in Communication Efficiency and MPI Parallel Efficiency.
So clearly, something in the communication of this model needs to be approved.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- Extrae
- Paraver



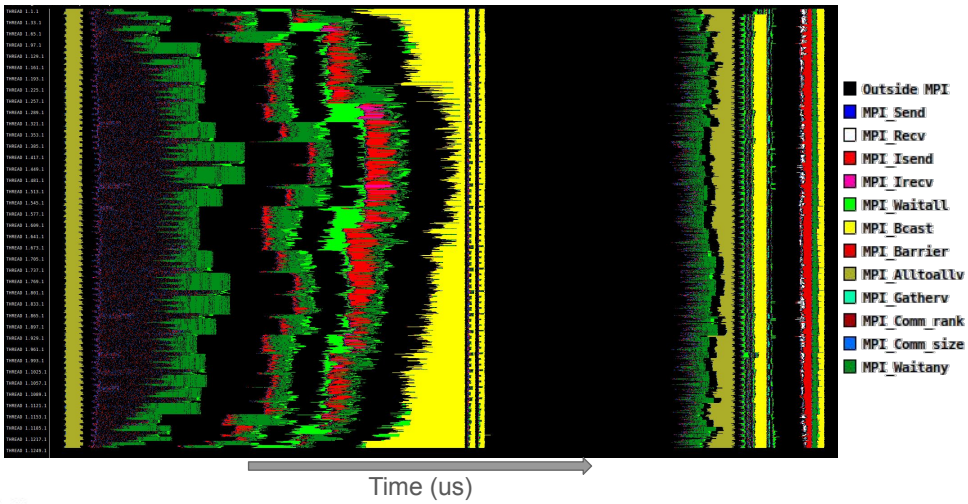
So we now know that we need to focus on the communication happening between the different nodes.
We again use our trace file of Extrae to visualize the communication that's happening.

That gives us an image like this.
It might look very scary to you, so let me explain what we are looking at.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- Extrae
- Paraver

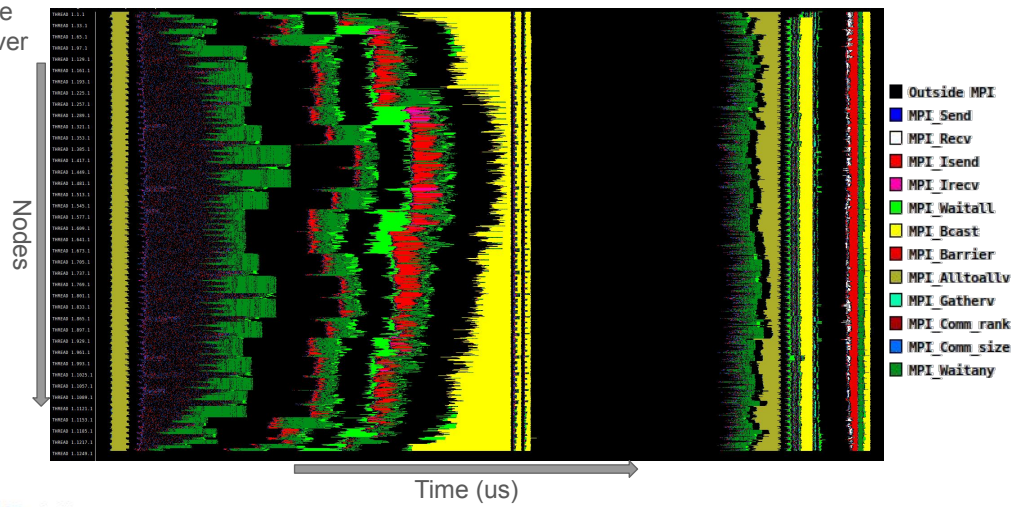


Over the horizontal axis, we have time.
It is basically a timeline what we are looking at.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- Extrae
- Paraver

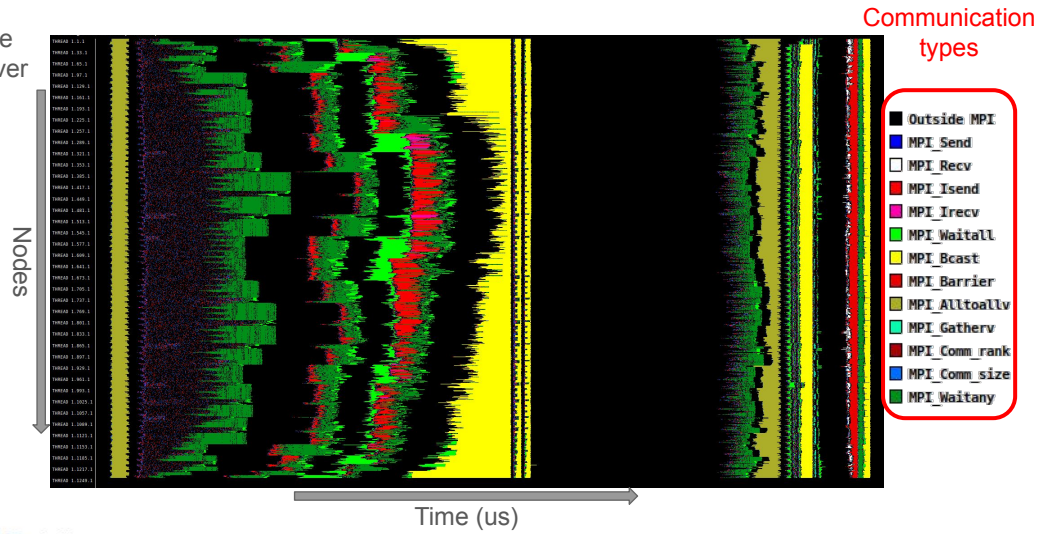


Then over the vertical axis, we have the different threads.
These are basically individual workers that are grouped per node, as in our HPC101 example.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- Extrae
- Paraver

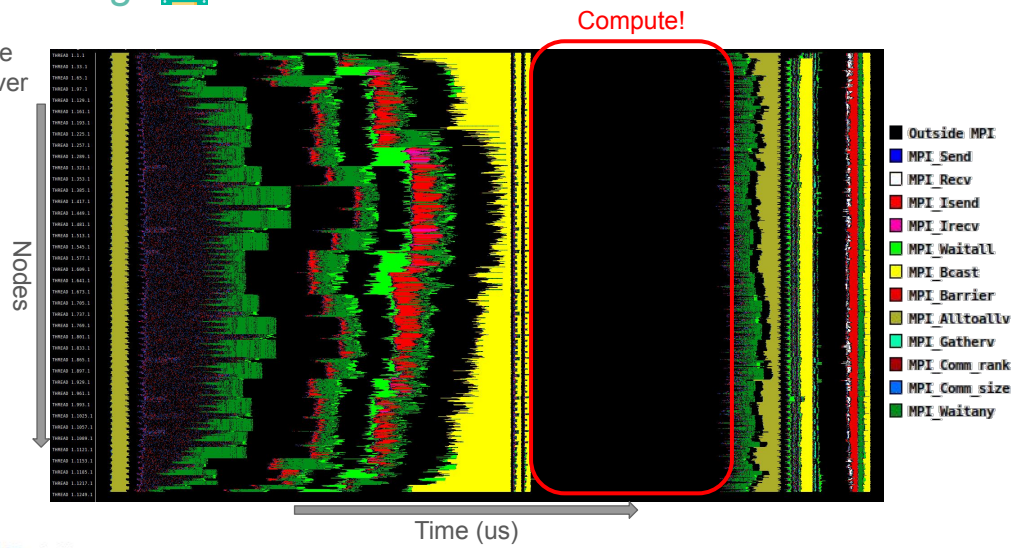


The different colors that you see are the different communication types that exist.
MPI is the protocol for sending data over a supercomputer.
So you can see that the Blue color corresponds to a send, while the white color corresponds to a receive, etc.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

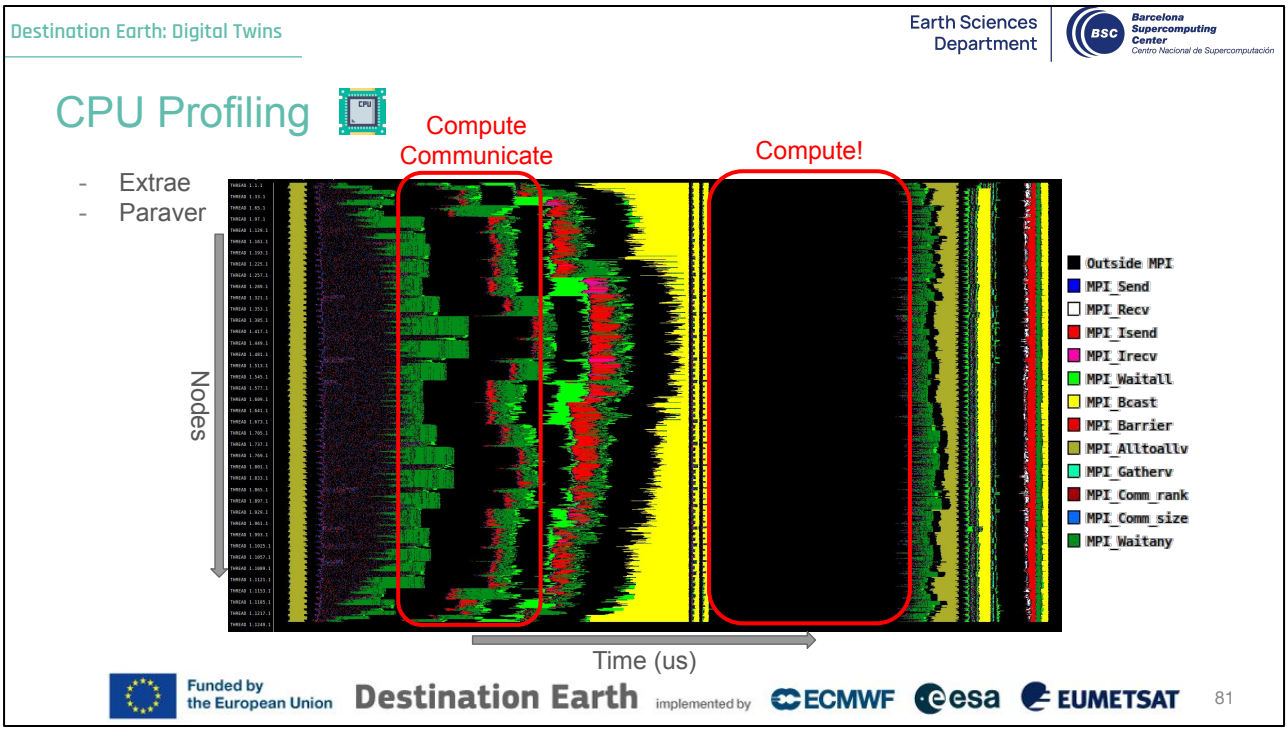
CPU Profiling

- Extrae
- Paraver



The black part is time spend outside of MPI, in other words, time where we compute or wait!

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391



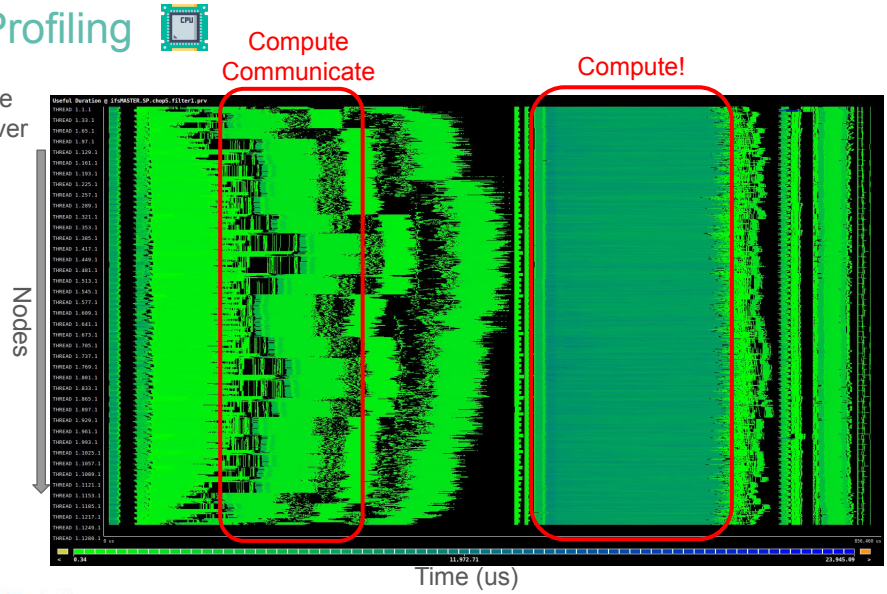
On the left we also see a very empty piece, where there is still some communication happening.
So here we have a mix of Compute, Communicate, and waiting.

This becomes even more clear if we switch our view to something called "Useful duration".

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- Extrae
- Paraver

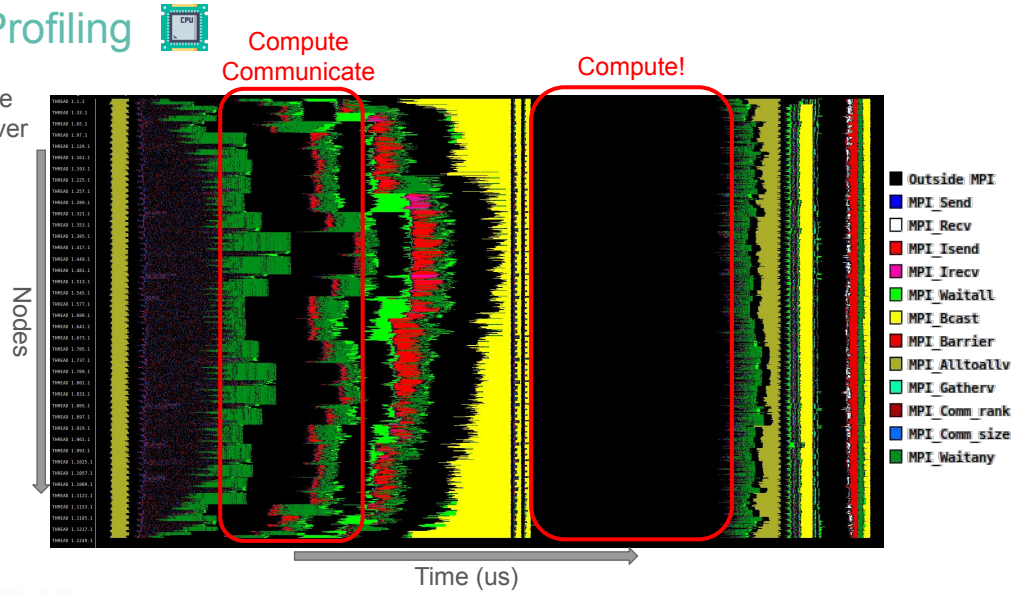


Here you have a scale going from green to blue, where the color corresponds to how much computation there is.
So you can see that the empty black part on the right is indeed mostly compute.
And on the right side we have an alternation between compute and communicate.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- Extrae
- Paraver

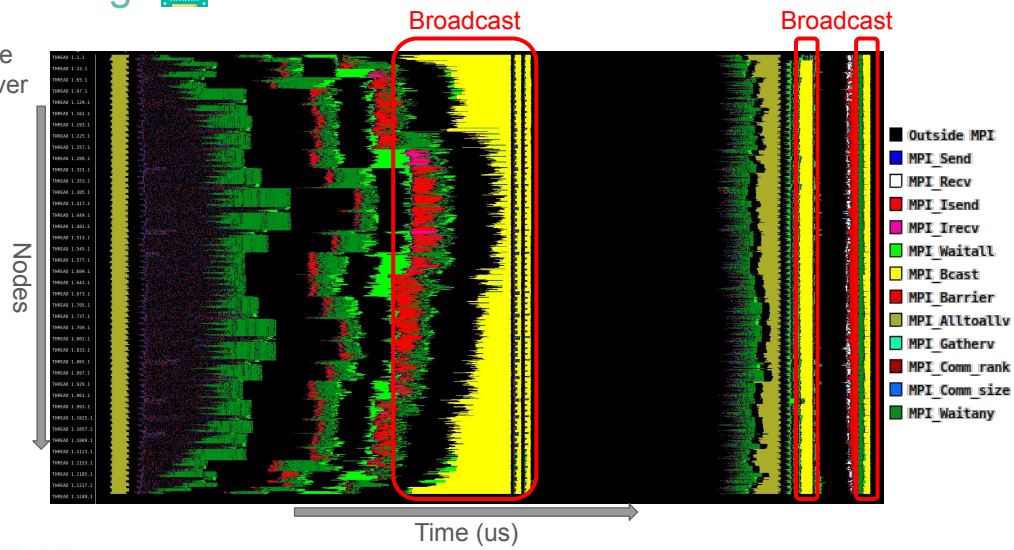


Going back to the communication view.
There are certain communication types that kill performance.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

CPU Profiling

- Extrae
- Paraver

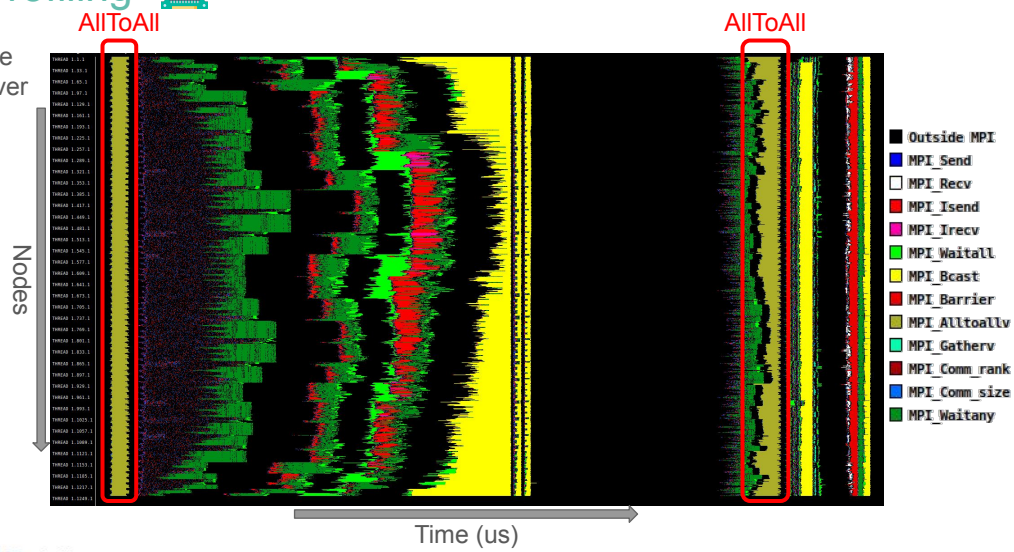


The Broadcast communication type is one of those.
In this case, some nodes send data which has to be received by all nodes before the process can continue.
Thus results in blocking communication, which makes nodes wait for each other instead of computing the problem.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

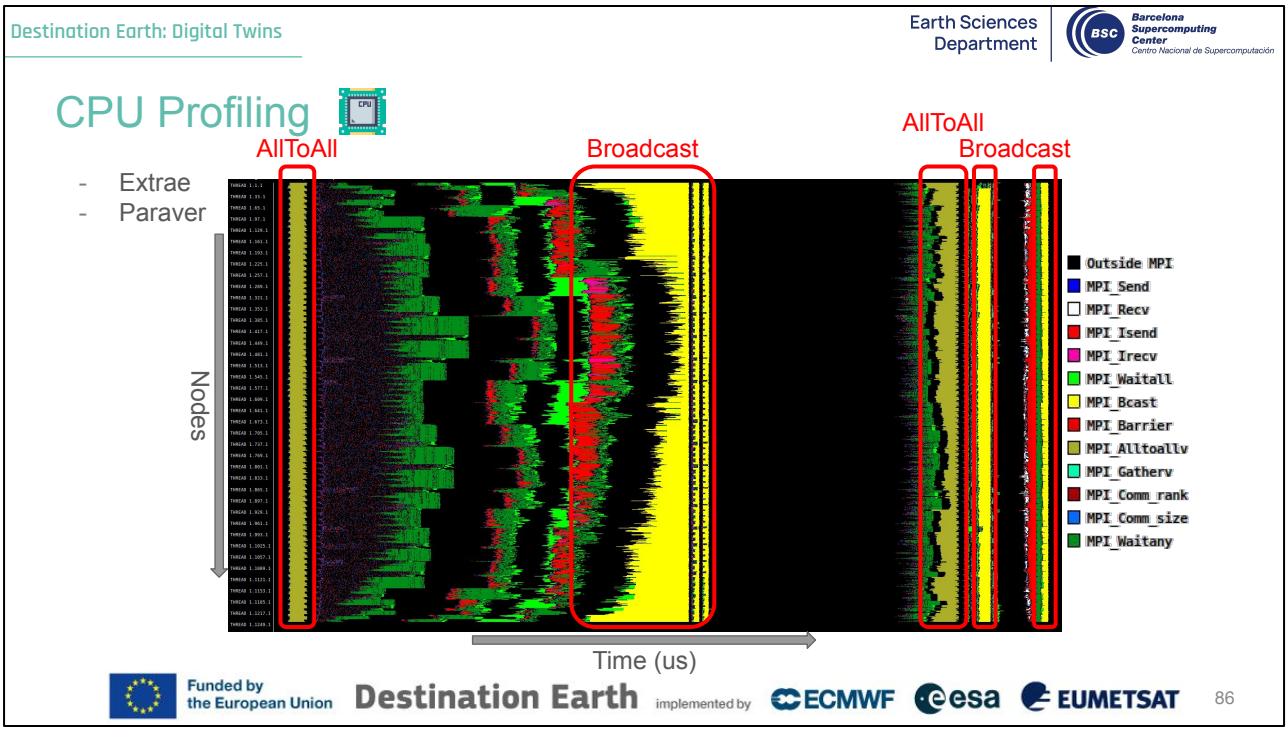
CPU Profiling

- Extrae
- Paraver



The AllToAll communication type has exactly the same problem, where the only difference is that all nodes are sending now as well.
This as well cause all nodes to wait for each other.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

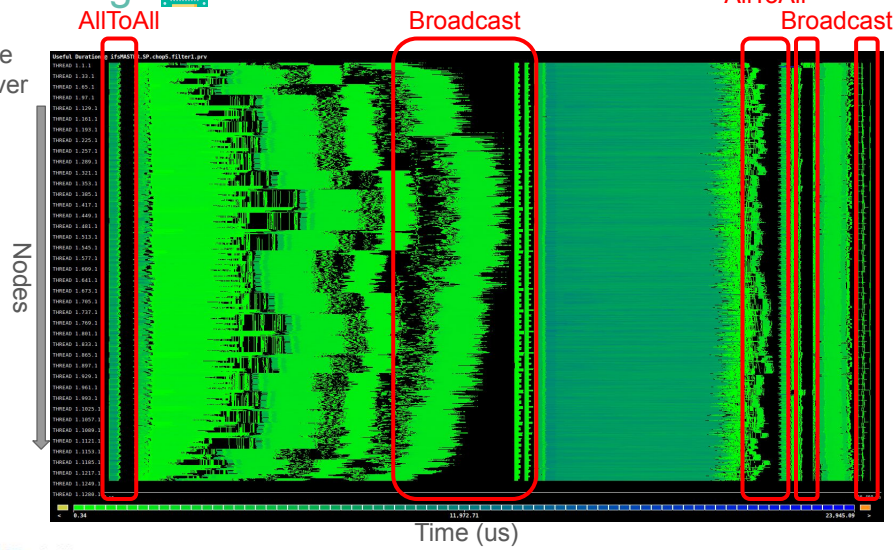


So take a close look at these yellow bits in the timeline.
When we switch to the useful duration view, where we see computation, you will see that nothing happens there.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

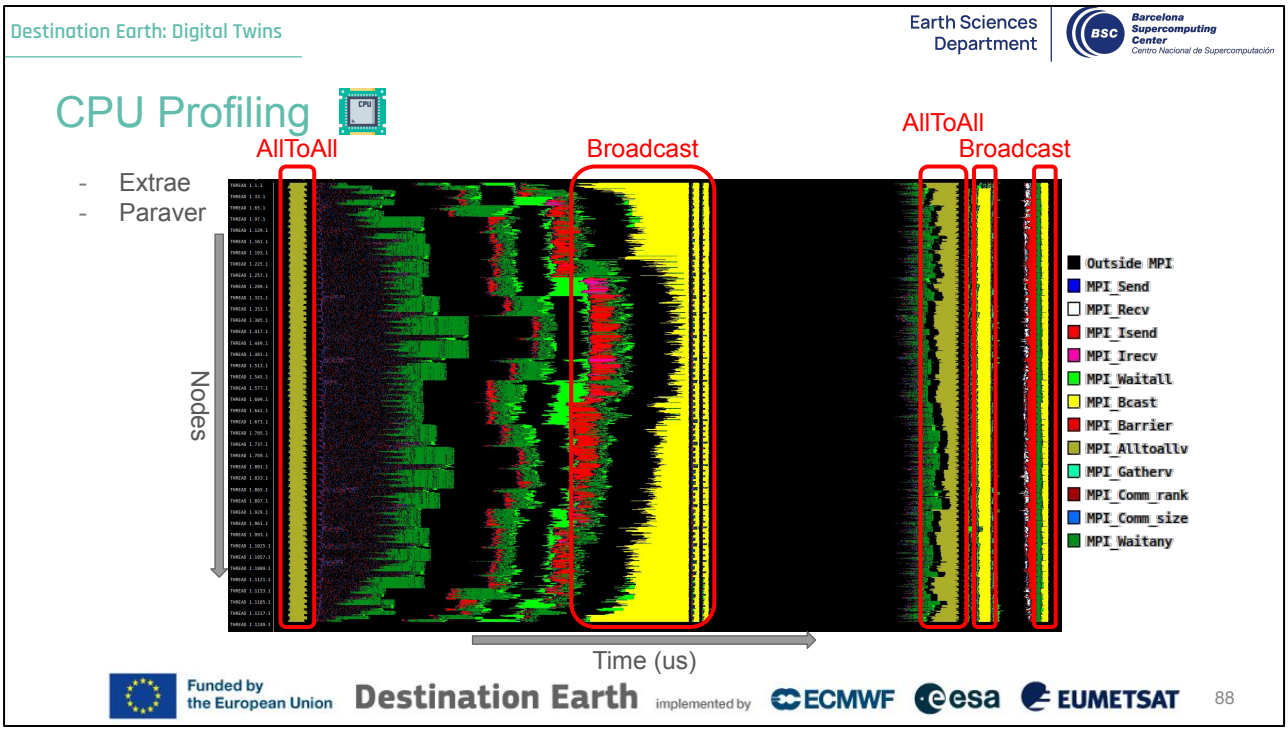
CPU Profiling

- Extrae
- Paraver



So here you can see that these communication types completely kill performance.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391



Thus these are patterns that we look out for when analyzing a timestep of a model.

CPU Icon:
https://www.flaticon.com/free-icon/cpu_984391

GPU Profiling

Platform matters!

Okay, so that was how we detect communication inefficiencies for CPUs.
Let's take a look at what we can do for GPUs.
First thing you need to know: platform matters!

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

GPU Profiling

Platform matters!

GPU

It matters what type GPU you have.
Generally you have NVIDIA, AMD, or Intel.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742
NVIDIA:
https://en.wikipedia.org/wiki/CUDA#/media/File:Nvidia_CUDA_Logo.jpg
AMD:
https://github.com/ROCm

GPU Profiling

Platform matters!

GPU

Compiler

Then it also matters what compiler you have,
since compilers can output debug information, but all do it in a different way.
Some examples here are OpenMPI, GNU, or Cray.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742
NVIDIA:
https://en.wikipedia.org/wiki/CUDA#/media/File:Nvidia_CUDA_Logo.jpg
AMD:
https://github.com/ROCm
Intel:
https://es.wikipedia.org/wiki/Archivo:Intel_logo_%282006-2020%29.svg
OpenMPI:
https://www.open-mpi.org/doc/v3.1/man1/mpifort.1.php
Cray:
https://insidehpc.com/wp-content/uploads/sites/2/2013/11/images.jpg
GNU:
https://es.wikipedia.org/wiki/GNU#/media/Archivo:Heckert_GNU_white.svg

GPU Profiling

Platform matters!

GPU

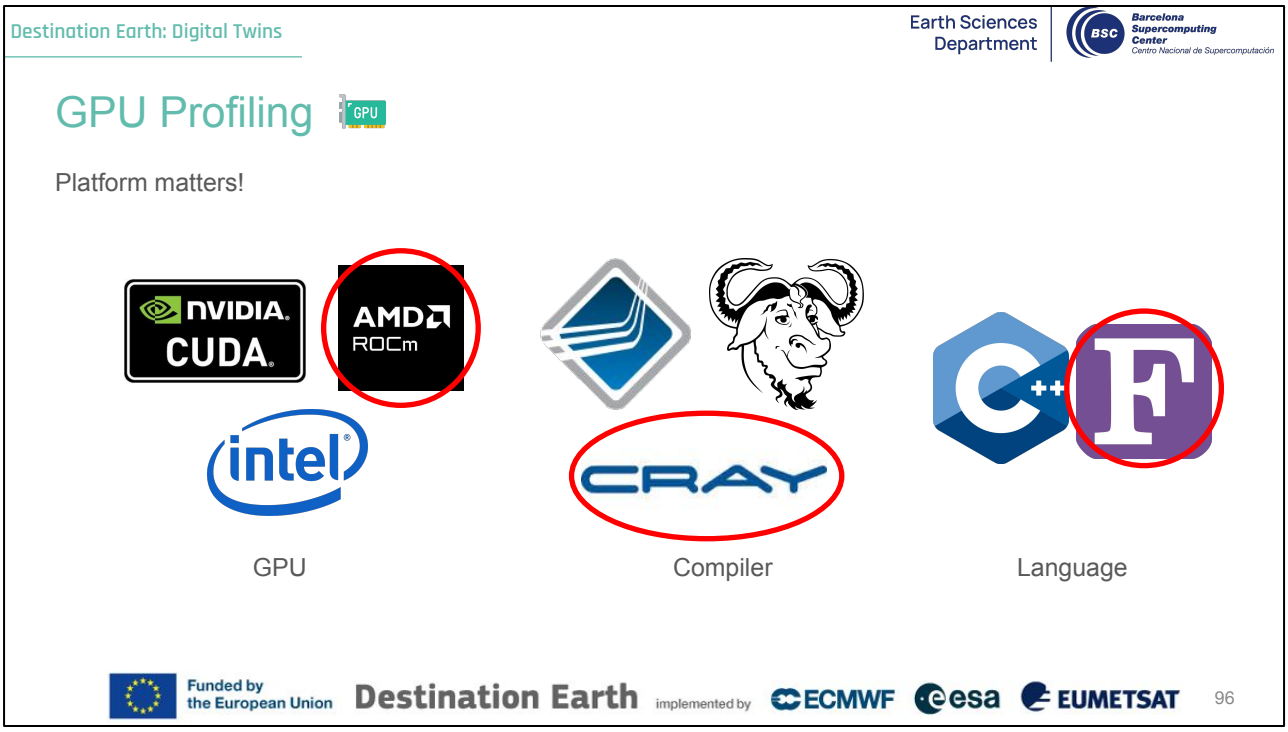
Compiler

Language

And lastly, it also matters what programming language your source code is in. Because Fortran for example, has less debug information during compilation time than when you compile C++.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742
NVIDIA:
https://en.wikipedia.org/wiki/CUDA#/media/File:Nvidia_CUDA_Logo.jpg
AMD:
https://github.com/ROCm
Intel:
https://es.wikipedia.org/wiki/Archivo:Intel_logo_%282006-2020%29.svg
OpenMPI:
https://www.open-mpi.org/doc/v3.1/man1/mpifort.1.php
Cray:
https://insidehpc.com/wp-content/uploads/sites/2/2013/11/images.jpg
GNU:
https://es.wikipedia.org/wiki/GNU#/media/Archivo:Heckert_GNU_white.svg
C++:
https://en.wikipedia.org/wiki/C%2B%2B#/media/File:ISO_C++_Logo.svg
Fortran:

https://commons.wikimedia.org/wiki/File:Fortran_logo.svg



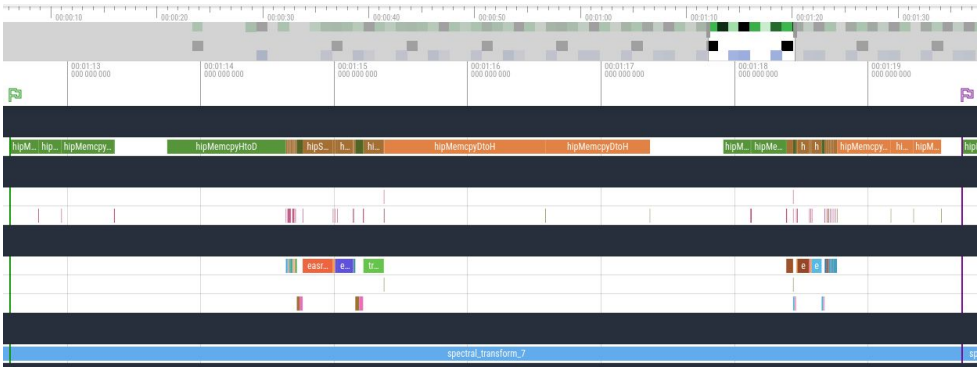
In this example, I'll take an example case for a report I made last year on the LUMI supercomputer in Finland.
This means we have AMD GPUs, the Cray FTN compiler, for our Fortran code base.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742
NVIDIA:
https://en.wikipedia.org/wiki/CUDA#/media/File:Nvidia_CUDA_Logo.jpg
AMD:
https://github.com/ROCm
Intel:
https://es.wikipedia.org/wiki/Archivo:Intel_logo_%282006-2020%29.svg
OpenMPI:
https://www.open-mpi.org/doc/v3.1/man1/mpifort.1.php
Cray:
https://insidehpc.com/wp-content/uploads/sites/2/2013/11/images.jpg
GNU:
https://es.wikipedia.org/wiki/GNU#/media/Archivo:Heckert_GNU_white.svg
C++:
https://en.wikipedia.org/wiki/C%2B%2B#/media/File:ISO_C++_Logo.svg
Fortran:

https://commons.wikimedia.org/wiki/File:Fortran_logo.svg

GPU Profiling

```
rocprof --hip-trace --roctx-trace -o ./rocprof/rocprof.csv $BIN
```



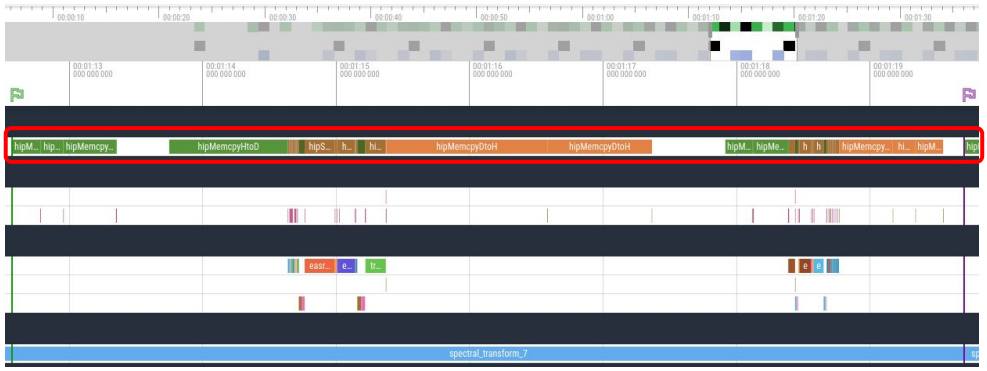
Perfetto visualization

With the AMD ROCm stack, you get a tool called Rocprof.
This is the most fundamental tool when profiling on AMD GPUs.
You can also collect a trace for a timeline, just like we did on CPU.
This is again 1 timestep of the code that we are analyzing, which is ecTrans, a spectral transformation code.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

GPU Profiling

```
rocprof --hip-trace --roctx-trace -o ./rocprof/rocprof.csv $BIN
```



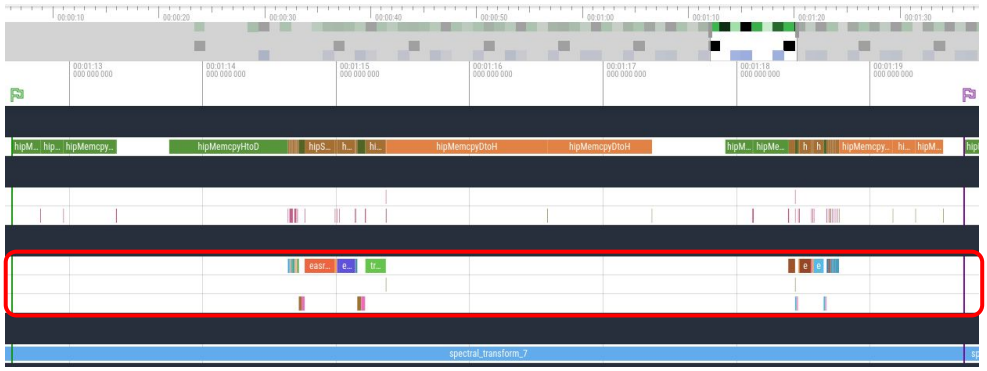
Perfetto visualization

On the top you have the memory transfers that occur between the CPUs and GPUs.
For those we don't know, a GPU is controlled from the CPU.
So the CPU will transfer the required data and instructions to the GPU, which executes them, and then the results are copied back.
These memory copies are very slow compared to the computation, and thus this is usually the main bottleneck for performance.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

GPU Profiling

```
rocprof --hip-trace --roctx-trace -o ./rocprof/rocprof.csv $BIN
```

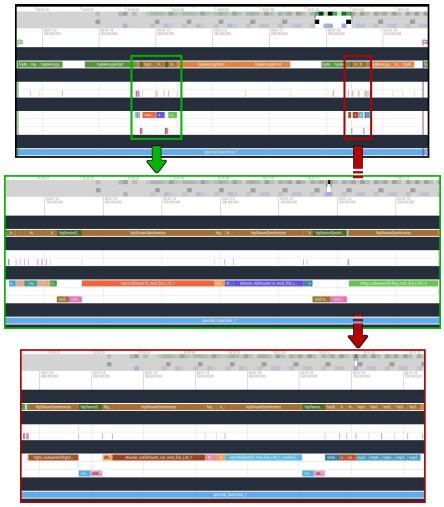


Perfetto visualization

Here you can see the GPU kernels, which are isolated programs that perform computations.
Ideally, you would like to spend most time executing kernels.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

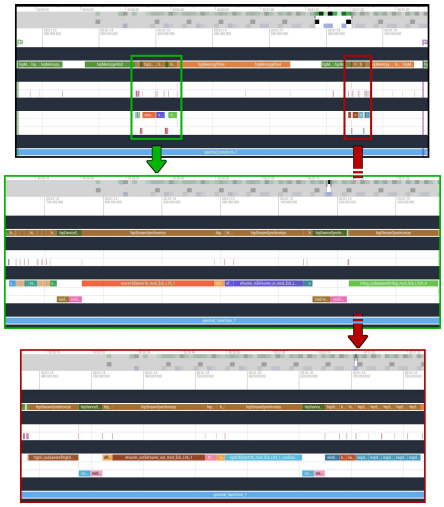
GPU Profiling

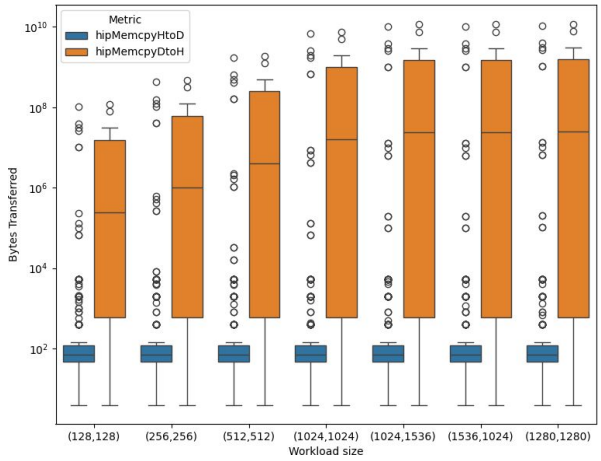


It might be a bit difficult to see, but here you have the two computation parts enlarged so it's easier to see the distribution between their runtimes.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

GPU Profiling



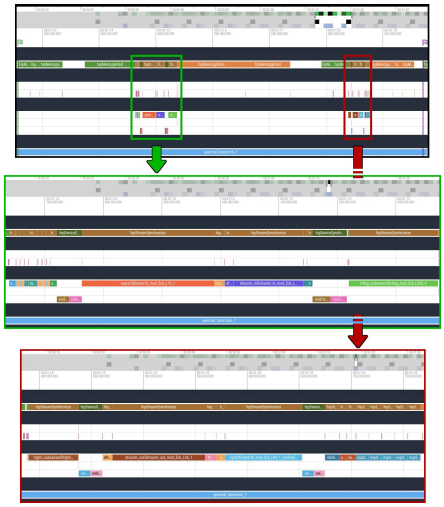


Since memory copies are usually a bottleneck, the first thing we do is parse this JSON into a boxplot as shown on the right side.
Here you see a column for each workload, which is basically how big the areas are that we are updating.
In blue you see the HostToDevice copies, while in orange you see the DeviceToHost copies.
Device in this case refers to the GPU, while the Host is the CPU.

The boxplots capture how big the transfers are that have been made.
And you can clearly see that the CPU to GPU transfers here are super small, while the GPU to CPU transfers are of okay size.
These super small transfers bring an overhead compared to large ones, since you keep calling the API.
Hence, this is causing quite a severe slowdown of the GPU performance, and thus our advice to our customer was to merge these transfers as much as possible.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

GPU Profiling



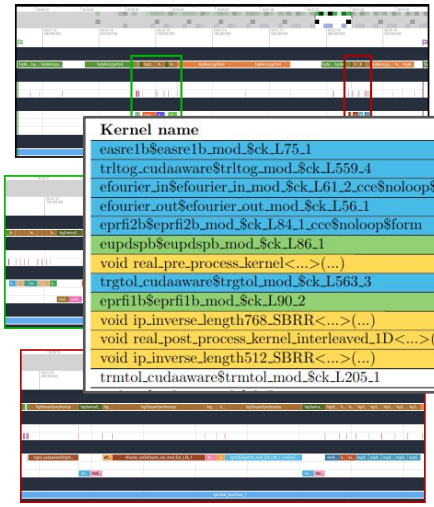
Kernel name	Calls	TotalDurationNs	AverageNs	Percentage
easre1b$easre1b_mod_$ck_L75_1	10	2 164 195 747	216 419 574	19,8255
trltog_cudaaware$trltog_mod_$ck_L559_4	10	1 531 345 585	153 134 558	14,0282
efourier_in$efourier_in_mod_$ck_L61_2_cce$noloop$form	10	1 198 170 052	119 817 005	10,9760
efourier_out$efourier_out_mod_$ck_L56_1	10	875 120 791	87 512 079	8,0167
eprfi2b$eprfi2b_mod_$ck_L84_1_cce$noloop$form	10	740 841 358	74 084 135	6,7866
eupdsph$eupdsph_mod_$ck_L86_1	60	612 827 387	10 213 789	5,6139
void real_pre_process_kernel<...>(...)	20	513 296 858	25 664 842	4,7021
trgtol_cudaaware$trgtol_mod_$ck_L563_3	10	479 085 073	47 908 507	4,3887
eprfi1b$eprfi1b_mod_$ck_L90_2	60	299 659 753	4 994 329	2,7451
void ip_inverse_length768_SBRR<...>(...)	10	282 009 617	28 200 961	2,5834
void real_post_process_kernel_interleaved_1D<...>(...)	20	226 225 138	11 311 256	2,0724
void ip_inverse_length512_SBRR<...>(...)	10	216 095 188	21 609 518	1,9796
trmtol_cudaaware$trmtol_mod_$ck_L205_1	10	178 045 797	17 804 579	1,6310
evdtuv$evdtuv_mod_$ck_L132_2	10	149 741 079	14 974 107	1,3717
efourier_in$efourier_in_mod_$ck_L58_3_cce$noloop$form	10	148 382 992	14 838 299	1,3593
eledir$eledir_mod_$ck_L98_1	10	139 369 528	13 936 952	1,2767
espnsde$espnsde_mod_$ck_L87_1	10	115 237 936	11 523 793	1,0557
eltinv$eltinv_mod_$ck_L136_2_cce$noloop$form	10	113 872 491	11 387 249	1,0431
trltom_cudaaware$trltom_mod_$ck_L218_1	10	110 770 720	11 077 072	1,0147
void ip_forward_length768_SBRR<...>(...)	10	110 712 645	11 071 264	1,0142
void ip_forward_length512_SBRR<...>(...)	10	110 086 079	11 008 607	1,0085
eftdir$eftdir_mod_$ck_L89_2_cce$noloop$form	10	94 305 395	9 430 539	0,8639
efsc$efsc_mod_$ck_L97_2	10	89 873 545	8 987 354	0,8233
euvtvd$euvtvd_mod_$ck_L108_2	10	85 680 570	8 568 057	0,7849
euvtvd$euvtvd_mod_$ck_L89_1	10	68 517 640	6 851 764	0,6277
evdtuv$evdtuv_mod_$ck_L113_1	10	66 336 511	6 633 651	0,6077
efsc$efsc_mod_$ck_L75_1	10	58 245 610	5 824 561	0,5336
eprfi1b$eprfi1b_mod_$ck_L85_3_cce$noloop$form	60	57 399 205	956 653	0,5258
eltdir$eltdir_mod_$ck_L157_3_cce$noloop$form	10	57 060 646	5 706 064	0,5227
eltdir$eltdir_mod_$ck_L156_2_cce$noloop$form	10	22 749 187	2 274 918	0,2084
<barrier packet>	20	753 461	37 673	0,0070
evdtuv$evdtuv_mod_$ck_L170_4	10	97 760	9776	0,009
euvtvd_comm$euvtvd_comm_mod_$ck_L106_2_cce$noloop$form	10	63 681	6368	0,0006
trgtol_cudaaware$trgtol_mod_$ck_L456_8	10	32 960	3296	0,0003
trgtol_cudaaware$trgtol_mod_$ck_L457_9	10	23 040	2304	0,0002

Table 8: Global overview of the ecTrans kernels for 10 iterations. The blue highlighted rows are the 6 user kernels which are analysed. The green highlighted rows have a smaller kernel granularity. The yellow highlighted rows are library calls.

Next, we can take a look at these kernels where the computations are happening.
The table here to the right shows all kernels in the timestep and how long they took.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

GPU Profiling



Kernel name	Calls	TotalDurationNs	AverageNs	Percentage
easre1b$easre1b_mod_$ck_L75_1	10	2 164 195 747	216 419 574	19,8255
trltog_cudaaware$trltog_mod_$ck_L559_4	10	1 531 345 585	153 134 558	14,0282
efourier_in$efourier_in_mod_$ck_L61_2_cce$noloop$form	10	1 198 170 052	119 817 005	10,9760
efourier_out$efourier_out_mod_$ck_L56_1	10	875 120 791	87 512 079	8,0167
eprfi2b$eprfi2b_mod_$ck_L84_1_cce$noloop$form	10	740 841 358	74 084 135	6,7866
eupdspb$eupdspb_mod_$ck_L86_1	60	612 827 387	10 213 789	5,6139
void real_pre_process_kernel<...>(...)	20	513 296 858	25 664 842	4,7021
trgtol_cudaaware$trgtol_mod_$ck_L563_3	10	479 085 073	47 908 507	4,3887
eprfi1b$eprfi1b_mod_$ck_L90_2	60	299 659 753	4 994 329	2,7451
void ip_inverse_length768_SBRR<...>(...)	10	282 009 617	28 200 961	2,5834
void real_post_process_kernel_interleaved_1D<...>(...)	20	226 225 138	11 311 256	2,0724
void ip_inverse_length512_SBRR<...>(...)	10	216 095 188	21 609 518	1,9796
trmtol_cudaaware$trmtol_mod_$ck_L205_1	10	178 045 797	17 804 579	1,6310
994 329				2,7451
8 200 961				2,5834
1 311 256				2,0724
1 609 518				1,9796
7 804 579				1,6310
4 974 107				1,3717
4 838 299				1,3593
3 936 952				1,2767
1 523 793				1,0557
1 387 249				1,0431
1 077 072				1,0147
1 071 264				1,0142
1 008 607				1,0085
9 430 539				0,8639
8 987 354				0,8233
8 568 057				0,7849
6 851 764				0,6277
6 633 651				0,6077
5 824 561				0,5336
956 653				0,5258
5 706 064				0,5227
2 274 918				0,2084
<barrier packet>	20	753 461	37 673	0,0070
evdtuv$evdtuv_mod_$ck_L170_4	10	97 760	9776	0,009
euvtvd_comm$euvtvd_comm_mod_$ck_L106_2_cce$noloop$form	10	63 681	6368	0,0006
trgtol_cudaaware$trgtol_mod_$ck_L456_8	10	32 960	3296	0,0003
trgtol_cudaaware$trgtol_mod_$ck_L457_9	10	23 040	2304	0,0002

Table 8: Global overview of the ecTrans kernels for 10 iterations. The blue highlighted rows are the 6 user kernels which are analysed. The green highlighted rows have a smaller kernel granularity. The yellow highlighted rows are library calls.

Let me enlarge the top rows so you can see better.
These numbers are for 10 time steps, so 10 calls means that there is 1 call per time step.
The blue kernels are the ones that we were investigating, since they looked the most promising as they took more than 3% of the computation time.

The green kernels also took significant time, but they had 60 calls, which means they are called 6 times.
This gives a bit of a skewed view, since you have 6 times the overhead of calling the function.
It could still be interesting to analyze, but with limited time we preferred to analyze the others.

And lastly we have the yellow kernels, which are system or library calls.
This is code that is called from another kernel and is not written by our client.
Hence, they cannot optimize these codes.

So let me show you how we would analyze a single kernel.
We could for example take a look at the top one, called easre1b with around 20% of computation time.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

GPU Profiling

```
rocprof -i rocprof_counters.txt -o ./rocprof/rocprof.csv $BIN
```

```
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,grd,wgr,lds,scr,vgpr,sgpr,fbar,sig,obj,GPUBusy,Wavefronts,VALUInsts,VFetchInsts,VWriteInsts,VALUUtilization,VALUBusy,WriteSize,L2CacheHit,MemUnitBusy,MemUnitStalled,LDSBankConflict
0,"eltinv$eltinv_mod_$ck_L136_2_cce$noloop$form.kd"0,0,0,26071,26071,1579087872,256,0,0,4,48,33664,0x0,0x14b05013c040,100.0000000000,24673248.0000000000,17.0000000000,0.0000000000,0.0000000000,100.0000000000,20.7725185968,12332528.0000000000,49.9978626693,77.1977044758,55.9370012636,0.0000000000
1,"eprfi1b$eprfi1b_mod_$ck_L85_3_cce$noloop$form.kd"0,0,2,26071,26071,157593600,256,0,0,4,72,38592,0x0,0x14b05013c100,100.0000000000,2462400.0000000000,20.0000000000,0.0000000000,0.0000000000,100.0000000000,24.2834884261,1227104.0000000000,49.9996649935,71.7003018579,49.2693704348,0.0000000000
2,"eprfi1b$eprfi1b_mod_$ck_L90_2.kd"0,0,4,26071,26071,56320,256,0,0,24,80,37696,0x0,0x14b05013c080,100.0000000000,880.0000000000,111907.6136363636,0.0000000000,0.0000000000,96.8019501699,9.6155911525,1008455.6562500000,81.3899031130,60.5181117611,43.6472872603,0.0000000000
3,"eprfi1b$eprfi1b_mod_$ck_L85_3_cce$noloop$form.kd"0,0,6,26071,26071,157593600,256,0,0,4,72,38592,0x0,0x14b0
...
```


To get performance metrics on a kernel, we have to collect so called hardware counters.
These are shown in the back of the header.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

GPU Profiling

```
rocprof -i rocprof_counters.txt -o ./rocprof/rocprof.csv $BIN
```

```
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,grd,wgr,lds,scr,vgpr,sgpr,fbar,sig,obj,GPUBusy,Wavefronts,VALUInsts,VFetchInsts,VWriteInsts,VALUUtilization,VALUBusy,WriteSize,L2CacheHit,MemUnitBusy,MemUnitStalled,LDSBankConflict
0,"eltinv$eltinv_mod_$ck_L136_2_cce$noloop$form.kd",0,0,0,26071,26071,1579087872,256,0,0,4,48,33664,0x0,0x14b05013c040,100.0000000000,24673248.0000000000,17.0000000000,0.0000000000,0.0000000000,100.0000000000,20.7725185968,12332528.0000000000,49.9978626693,77.1977044758,55.9370012636,0.0000000000
1,"eprfi1b$eprfi1b_mod_$ck_L85_3_cce$noloop$form.kd",0,0,2,26071,26071,157593600,256,0,0,4,72,38592,0x0,0x14b05013c100,100.0000000000,2462400.0000000000,20.0000000000,0.0000000000,0.0000000000,100.0000000000,24.2834884261,1227104.0000000000,49.9996649935,71.7003018579,49.2693704348,0.0000000000
2,"eprfi1b$eprfi1b_mod_$ck_L90_2.kd",0,0,4,26071,26071,56320,256,0,0,24,80,37696,0x0,0x14b05013c080,100.0000000000,880.0000000000,111907.6136363636,0.0000000000,0.0000000000,96.8019501699,9.6155911525,1008455.6562500000,81.3899031130,60.5181117611,43.6472872603,0.0000000000
3,"eprfi1b$eprfi1b_mod_$ck_L85_3_cce$noloop$form.kd",0,0,6,26071,26071,157593600,256,0,0,4,72,38592,0x0,0x14b0
...
```


To get performance metrics on a kernel, we have to collect so called hardware counters.
You write these in a file called rocprof_counters.txt, but they are also shown in the back of the header.
And so for each kernel, we collect the value of these counters, for each time step, and for different workloads.

There is no default visualization, and thus I have written my own matrix visualization of these counters for different workloads.

GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

GPU Profiling

GPU Modelfactors for easre1b$easre1b_mod_$ck_L75_1.kd.

	128x128	256x256	512x512	1024x1024
Device Load (% of device)				
Wavefronts	25.00	25.00	25.00	25.00
Occupancy (% of device)				
vgpr	7.81	7.81	7.81	7.81
sgpr	70.59	70.59	70.59	70.59
Memory Efficiency (% of 128x128)				
MemUnitBusy	96.58	94.87	97.16	91.46
L2CacheHit	88.93	55.59	22.10	10.81
MemUnitStalled	33.62	38.04	40.17	38.69
LDSBankConflict	0.00	0.00	0.00	0.00
Computational Efficiency (% of 128x128)				
VALUUtilization	100.00	100.00	100.00	100.00
VALUBusy	24.45	12.14	8.47	7.60

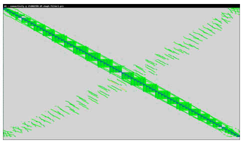
Relative load (%): 100, 80, 60, 40, 20, 0

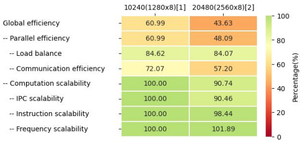
Here you can see that visualization, which is very much like the CPU Modelfactors of Paraver.
The columns are different workloads, or grid sizes, which were also used for the memory plot.
The rows are the hardware counters that we test for.

I won't get into detail what each hardware counter resembles, but I'd say it's easy to see what the problem is with this kernel.
The L2CacheHit rate goes down from 88.93% to 10.81% when increasing the workload.
At the same time we see the VALUBusy metric go down from 24.45% to 7.60%.
So clearly, we have a lot of memory loads that make the computational ALU units wait.
Hence, our advice is to optimize the memory layout for this kernel.
We expect quite an improvement, since it took close to 20% of the computational time.

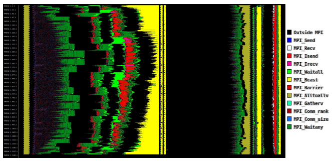
GPU Icon:
https://www.flaticon.com/free-icon/gpu_4617742

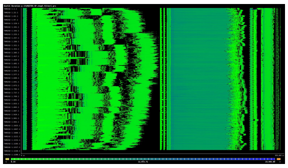
Performance profiling

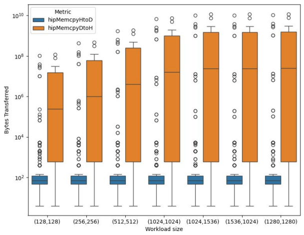


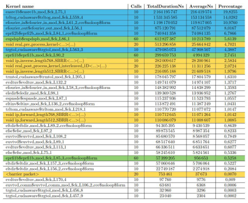


	10240(1280x8)[1]	20480(2560x8)[2]
Global efficiency	60.99	43.63
-- Parallel efficiency	60.99	48.09
-- Load balance	84.62	84.07
-- Communication efficiency	72.07	57.20
-- Computation scalability	100.00	90.74
-- IPC scalability	100.00	90.46
-- Instruction scalability	100.00	98.44
-- Frequency scalability	100.00	101.89



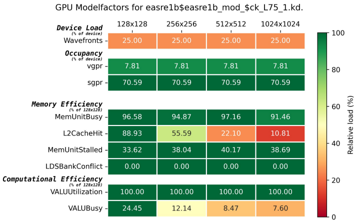






Kernel name	Calls	TotalDurationNs	AverageNs	Percentage
easre1b$easre1b_mod_$ck_L75_1	10	2 164 195 747	216 419 574	19,8255
trltog_cudaaware$trltog_mod_$ck_L559_4	10	1 531 345 585	153 134 558	14,0282
efourier_in$efourier_in_mod_$ck_L61_2_cce$noloop$form	10	1 198 170 052	119 817 005	10,9760
efourier_out$efourier_out_mod_$ck_L56_1	10	875 120 791	87 512 079	8,0167
eprfi2b$eprfi2b_mod_$ck_L84_1_cce$noloop$form	10	740 841 358	74 084 135	6,7866
eupdspb$eupdspb_mod_$ck_L86_1	60	612 827 387	10 213 789	5,6139
void real_pre_process_kernel<...>(...)	20	513 296 858	25 664 842	4,7021
trgtol_cudaaware$trgtol_mod_$ck_L563_3	10	479 085 073	47 908 507	4,3887
eprfi1b$eprfi1b_mod_$ck_L90_2	60	299 659 753	4 994 329	2,7451
void ip_inverse_length768_SBRR<...>(...)	10	282 009 617	28 200 961	2,5834
void real_post_process_kernel_interleaved_1D<...>(...)	20	226 225 138	11 311 256	2,0724
void ip_inverse_length512_SBRR<...>(...)	10	216 095 188	21 609 518	1,9796
trmtol_cudaaware$trmtol_mod_$ck_L205_1	10	178 045 797	17 804 579	1,6310
evdtuv$evdtuv_mod_$ck_L132_2	10	149 741 079	14 974 107	1,3717
efourier_in$efourier_in_mod_$ck_L58_3_cce$noloop$form	10	148 382 992	14 838 299	1,3593
eledir$eledir_mod_$ck_L98_1	10	139 369 528	13 936 952	1,2767
espnsde$espnsde_mod_$ck_L87_1	10	115 237 936	11 523 793	1,0557
eltinv$eltinv_mod_$ck_L136_2_cce$noloop$form	10	113 872 491	11 387 249	1,0431
trltom_cudaaware$trltom_mod_$ck_L218_1	10	110 770 720	11 077 072	1,0147
void ip_forward_length768_SBRR<...>(...)	10	110 712 645	11 071 264	1,0142
void ip_forward_length512_SBRR<...>(...)	10	110 086 079	11 008 607	1,0085
eftdir$eftdir_mod_$ck_L89_2_cce$noloop$form	10	94 305 395	9 430 539	0,8639
efsc$efsc_mod_$ck_L97_2	10	89 873 545	8 987 354	0,8233
euvtvd$euvtvd_mod_$ck_L108_2	10	85 680 570	8 568 057	0,7849
euvtvd$euvtvd_mod_$ck_L89_1	10	68 517 640	6 851 764	0,6277
evdtuv$evdtuv_mod_$ck_L113_1	10	66 336 511	6 633 651	0,6077
efsc$efsc_mod_$ck_L75_1	10	58 245 610	5 824 561	0,5336
eprfi1b$eprfi1b_mod_$ck_L85_3_cce$noloop$form	60	57 399 205	956 653	0,5258
eltdir$eltdir_mod_$ck_L157_3_cce$noloop$form	10	57 060 646	5 706 064	0,5227
eltdir$eltdir_mod_$ck_L156_2_cce$noloop$form	10	22 749 187	2 274 918	0,2084
<barrier packet>	20	753 461	37 673	0,0070
evdtuv$evdtuv_mod_$ck_L170_4	10	97 760	9776	0,009
euvtvd_comm$euvtvd_comm_mod_$ck_L106_2_cce$noloop$form	10	63 681	6368	0,0006
trgtol_cudaaware$trgtol_mod_$ck_L456_8	10	32 960	3296	0,0003
trgtol_cudaaware$trgtol_mod_$ck_L457_9	10	23 040	2304	0,0002

Table 8: Global overview of the ecTrans kernels for 10 iterations. The blue highlighted rows are the 6 user kernels which are analysed. The green highlighted rows have a smaller kernel granularity. The yellow highlighted rows are library calls.



GPU Modelfactors for easre1b$easre1b_mod_$ck_L75_1.kd.

	128x128	256x256	512x512	1024x1024
Device Load *(% of device)*				
Wavefronts	25.00	25.00	25.00	25.00
Occupancy *(% of device)*				
vgpr	7.81	7.81	7.81	7.81
sgpr	70.59	70.59	70.59	70.59
Memory Efficiency *(% of 128x128)*				
MemUnitBusy	96.58	94.87	97.16	91.46
L2CacheHit	88.93	55.59	22.10	10.81
MemUnitStalled	33.62	38.04	40.17	38.69
LDSBankConflict	0.00	0.00	0.00	0.00
Computational Efficiency *(% of 128x128)*				
VALUUtilization	100.00	100.00	100.00	100.00
VALUBusy	24.45	12.14	8.47	7.60

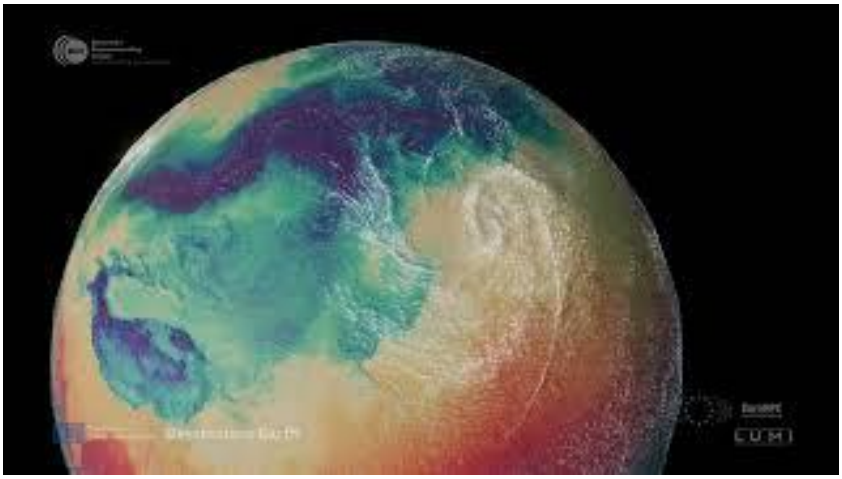
And thus you can see, we have many different ways to visualize the performance of a model.
In this presentation we only covered the tip of the iceberg, but you can go much deeper into questions like why a model is computing more/less in certain regions.

The earth is still alive..

That sums up my quick overview of how to detect
Communication inefficiencies on the CPU
And memory issues on the GPU
Of course, there is much much more to explain, but I hope this got you a bit triggered to think more about the performance of you code when you program.

To close my talk, I'd like to end on a positive note.
Climate change is getting worse and worse, but the Earth is still trying to adapt.
And many of the important climate systems still live.
So I want to show you our latest visualizations of the earth, which shows many climate related phenomena.

The earth is still alive..



https://www.youtube.com/watch?v=hhJW0RBq0hA&list=PLbABsNMD2jhy5zsINz9VIuOzVsrM0_vwu

0:00-0:45: clouds forming and moving + hurricanes
1:12-3:00: earth heightmap, sea temperature streams,
5:10-6:30: wind map, to pressure + temperature map, heart beat from day night

Total: 45+108+80=233s = 3:53m

That was all from my side.
I hope you learned something about climate modellen or performance engineering.
Here you have my contact information in case you want to contact me later, but please ask any questions you have now as well.
Thank you!