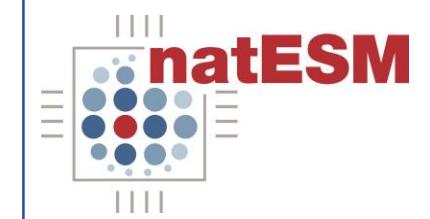


Generic concepts of ICON ComIn

F. Prill & ICON ComIn project team



DWD, DLR, DKRZ | natESM Tech Training | Hamburg | 17 July 2024



Intro: Plugin mechanism for ICON

Demo script that runs alongside the ICON simulation:

```
import comin ←  
import numpy as np  
  
comin.var_request_add("my_var", 1), False)  
  
@comin.register_callback(←  
    comin.EP_SECONDARY_CONSTRUCTOR)  
def simple_python_constructor():  
    global pres, my_var  
    print("simple_python_constructor called!")  
    pres = comin.var_get(  
        [comin.EP_ATM_WRITE_OUTPUT_BEFORE],  
        ("pres", 1))  
    my_var = comin.var_get(  
        [comin.EP_ATM_WRITE_OUTPUT_BEFORE],  
        ("my_var", 1))
```



```
@comin.register_callback(←  
    comin.EP_ATM_WRITE_OUTPUT_BEFORE)  
def simple_python_diagfct():  
    np.asarray(my_var)[:] = 42.
```

```
@comin.register_callback(comin.EP_DESTRUCTOR)  
def simple_python_destructor():  
    print("simple_python_destructor called!")
```



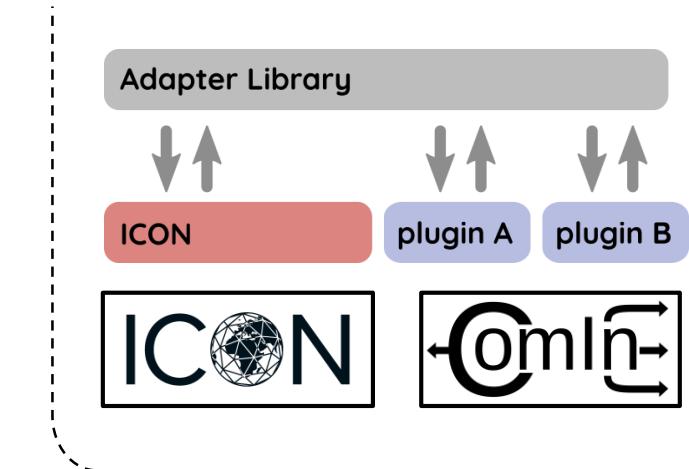
ComIn = ICON Community Interface

The community interface

- connects 3rd party modules to the ICON host model
- regulates the access and creation of model variables
- plugin functions are called at pre-defined events

Shared libraries for Fortran or C/C++ plugins.

Python plugins run without compilation process.



Project started 2022 as a collaboration between **DWD**, **DLR-IPA** and **DKRZ**.

ComIn is part of the Open Source Release of ICON 2024.01 ^[1].

Note: Similar solutions exist for the NOAA models or the IFS ^[2].



[1] <https://gitlab.dkrz.de/icon/icon-model>

[2] IFS Plume: <https://github.com/ecmwf/plume>

Running an existing plugin



Plugging into ICON's control flow

About 40 entry points

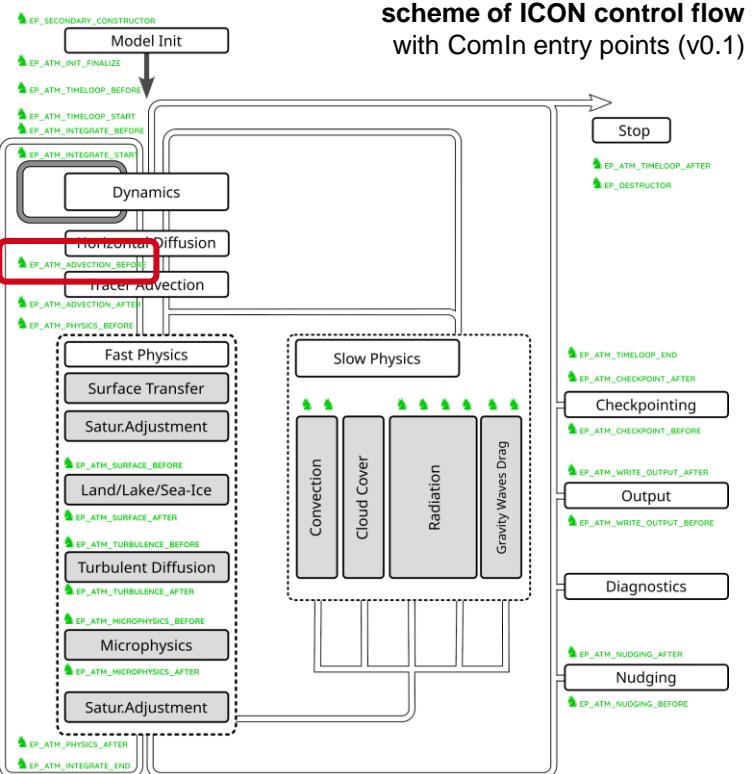
- Ex.: **EP_ATM_WRITE_OUTPUT_BEFORE**

“before/after” nomenclature

EP_<COMP>_<PROCESS | LOOP>_
[BEFORE | AFTER | START | END]

ComIn
entry points

- ICON host model: look out for
CALL icon_call_callback(...)
- dependent on namelist settings and
program flow
- preliminary and easily expandable!
Feedback welcome!

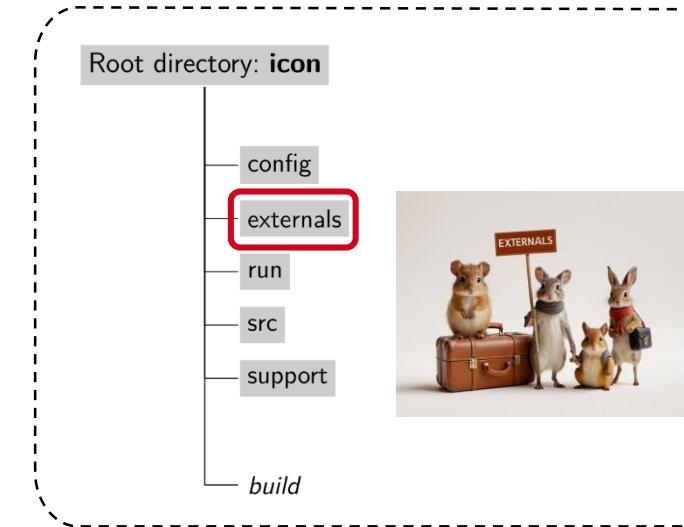


How to enable ICON ComIn

ComIn is a switchable external ICON package contained in the ICON release <https://gitlab.dkrz.de/icon/icon-model>.

```
../config/dkrz/levante.gcc --enable-comin
```

- Alternatively, you can download a standalone version. ComIn releases published independently from ICON [1].
- The [Python adapter](#) itself is implemented as a ComIn plugin written in C++. It comes together with the ComIn library and embeds the Python interpreter.



User guide, technical description ... <https://icon-comin.gitlab-pages.dkrz.de/comin>



[1] <https://gitlab.dkrz.de/icon-comin/comin/-/releases>

configure --enable-comin ... and then what?

Register your plugin in the host model via namelist.

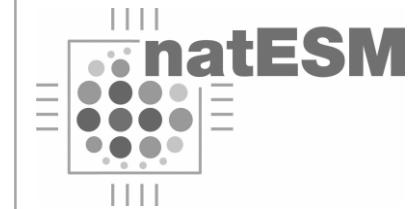


```
&comin_nml
    plugin_list(1)%name          = "comin_plugin"
    plugin_list(1)%plugin_library = "libpython_adapter.so"
    plugin_list(1)%options       = "comin_plugin.py"
/
    filename of the Python script is passed as an option
```

*Optional: non-standard name of constructor, MPI communicator,
user-specified plugin arguments (character string)*



Developing a plugin



primary constructor / callback function(s) / descriptive data structures / variable access



Building blocks ... Primary constructor

Primary constructor: registers the plugin and additional variables.

- Note: In Fortran or C/C++ plugins, the primary constructor is a separate subroutine; for Python this is simply the main body.
- API functions are exposed in the Python module `comin`, needs to be imported.

```
import comin  
  
var_descriptor = ("comin_process_id", jg)  
  
comin.var_request_add(var_descriptor, lmodexclusive=False)  
  
comin.metadata_set(var_descriptor, zaxis_id = comin.COMIN_ZAXIS_2D)
```

variables are identified
by name + domain ID



main purpose of the primary constructor:
request additional variables



Building blocks ... Secondary constructor

By the term **secondary constructor** we denote an entry point at the end of ICON's initialization. That's the first time when you can obtain readable and writable pointers to the ICON data fields.

```
myvar = comin.var_get([comin.EP_ATM_WRITE_OUTPUT_BEFORE],  
                      ("comin_process_id", jg),  
                      flag=comin.COMIN_FLAG_WRITE)
```

this is a variable handle!
see next slide.

this tells ComIn about the context,
ie. the entry point where we use the variable

again, ComIn identifies variables
by name + domain ID



Descriptive data structures contain information on the ICON setup and the simulation:

```
domain = comin.descrdata_get_domain(jg)  
clon = np.asarray(domain.cells.clon)
```



Building blocks ... Callback function(s)

Callbacks for specific entry points are realized as custom Python functions.

- registered with ComIn by adding a **function decorator**
- note: the secondary constructor is a callback like any other

```
@comin.register_callback(comin.EP_ATM_WRITE_OUTPUT_BEFORE)
def simple_python_callbackfct():
    myvar_array = np.asarray(myvar)      variable handle from the previous slide
```



Access to the variable data is provided via NumPy/CuPy objects.

Building blocks: Summary

```
import comin
import numpy as np
primary constructor:
request additional ICON variable

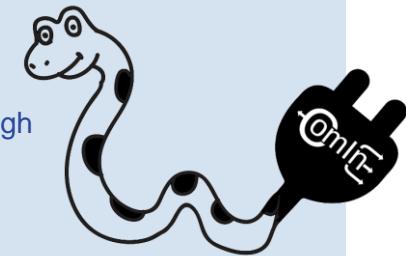
comin.var_request_add(("my_var", 1), False)
```

```
@comin.register_callback(←
    comin.EP_SECONDARY_CONSTRUCTOR)
def simple_python_constructor():
    global pres, my_var
    print("simple_python_constructor called!")
    pres = comin.var_get(
        [comin.EP_ATM_WRITE_OUTPUT_BEFORE],
        ("pres", 1))
    my_var = comin.var_get(
        [comin.EP_ATM_WRITE_OUTPUT_BEFORE],
        ("my_var", 1))
secondary constructor: get variable handles
```

callbacks: register function callbacks through Python decorators

```
@comin.register_callback(←
    comin.EP_ATM_WRITE_OUTPUT_BEFORE)
def simple_python_diagfct():
    np.asarray(my_var)[:] = 42.
```

```
@comin.register_callback(comin.EP_DESTRUCTOR)
def simple_python_destructor():
    print("simple_python_destructor called!")
```



See the ComIn release code for more demo plugins (Fortran, C/C++, Python).



Building blocks: Summary

```
import comin  
import numpy as np
```

primary constructor:
request additional ICON variable

```
@comin.register_callback(↔
    comin.EP_SECONDARY_CONSTRUCTOR)
def simple_python_constructor():
    global pres, my_var
    print("simple_python_constructor")
    pres = comin.var_get(
        [comin.EP_ATM_WRITE_OUTPUT_
         ("pres", 1)])
    my_var = comin.var_get(
        [comin.EP_ATM_WRITE_OUTPUT_
         ("my_var", 1)])
```

Compare this to the “heavy-weight” addition of new variables to ICON.

... lots of Fortran **CALLS** and loops

ComIn plugins are lightweight.

secondary constructor: get variable here

callbacks: register function callbacks through Python decorators

Accomplish register callback /



ICON Tutorial 2024 Section 9.4:
Programming ICON / Implementing diagnostics

8. Programming in ICON	
ICON Model Tutorial	8.3 Implementing Own Diagnostics
The code in the module <code>src/parallel_infrastructure/api.koml</code> (Fig. 8.6) probably the most important part of the ICON code is <code>g_koml_error</code> . It is related to the message of error and can be used to report errors in the system or to report errors in the work process.	<p>The <code>g_koml_error</code> function is a global variable, whose value may change as we run by calling <code>get_g_koml_error()</code>. On a user level, the work process can use this variable to check its own validation. It is the mechanism used by the <code>exchange_data</code> function to validate content.</p>
A final result is related to the everlasting lifetime of the static <code>log</code> (newer) object. This is done by the <code>log</code> function, which is part of the communication process <code>api.koml</code> , which is the MTP communication component of PNs that are running in the model. A typical use of the <code>log</code> function message is shown below:	<pre>if (msg == "log") { log("Hello world!"); }</pre>
With this message, the PN will print the message on PNs 1-2.	<p>Of course, there always exists an auxiliary message manager () in ICON, where each process is what is selected manually above.</p>
The message <code>"log"</code> is a static variable of the module <code>exchange_data</code> and prints the message to PN 0. It takes the value's reference name as an additional argument.	<pre>static variable "log" of module exchange_data</pre>
9.3 Implementing Own Diagnostics	
A diagnostic mechanism is used to monitor the ICON model and implement one's own diagnostic tools directly by a developer in his/her icon file. Here, we will keep things as simple and short as possible.	<p>Afterwards, the dependency generation of ICON automatically determines which parts of the code need to be checked and which parts do not need to be automatically included in the compilation process. This, when creating a new module, is done by the developer and can be done with <code>ICOM</code> as a tool (see module <code>stable_and_independent.koml</code>).</p>
<p>Adding new statics: The <code>ICON</code> kernel needs statics defined in programs and diagnostic files. This global section now has the task of defining (declaration and assignment) the field's variables, e.g., the <code>error</code> declaration, which is a static variable that will be requiring a new static variable. The static variable <code>g_koml_error</code> is declared in the <code>stable_and_independent.koml</code> header module and will therefore always stay in step.</p>	<pre>static variable "g_koml_error" of module stable_and_independent = 0;</pre>
<p>Final statics: The static variable <code>g_koml_error</code> to which we can assign our variable. For the sake of simplicity, we choose to create a variable variable <code>list</code>, defined in the module <code>exchange_data.koml</code>.</p>	<pre>static variable "list" of module exchange_data = null;</pre>
<p>Adding new statics: <code>ICON</code> kernel needs statics defined in programs and diagnostic files. This global section now has the task of defining (declaration and assignment) the field's variables, e.g., the <code>error</code> declaration, which is a static variable that will be requiring a new static variable. The static variable <code>g_koml_error</code> is declared in the <code>stable_and_independent.koml</code> header module and will therefore always stay in step.</p>	<pre>static variable "g_koml_error" of module stable_and_independent = 0;</pre>
<p>Final statics: The static variable <code>g_koml_error</code> to which we can assign our variable. For the sake of simplicity, we choose to create a variable variable <code>list</code>, defined in the module <code>exchange_data.koml</code>.</p>	<pre>static variable "list" of module exchange_data = null;</pre>

Fortran and C/C++ API

The previous slides focused on Python. But: ComIn also contains a Fortran and C/C++-API.

Fortran API	C/C++ API	Python API
comin_setup_get_verbosity_level	int comin_setup_get_verbosity_level()	comin.setup_get_verbosity_level
comin_current_get_ep	int comin_current_get_ep()	
comin_current_get_domain_id	int comin_current_get_domain_id()	comin.current_get_domain_id
comin_current_get_datetime	void comin_current_get_datetime(char const**,int*,int*)	comin.comin_current_get_datetime
comin_current_get_plugin_info	int comin_current_get_plugin_id()	comin.current_get_plugin_info
	void comin_current_get_plugin_name(char const **, int*, int*)	
	void comin_current_get_plugin_options(char const **,int*,int*)	
	void comin_current_get_plugin_comm(char const **,int*,int*)	
comin_parallel_get_plugin_mpi_comm	int comin_parallel_get_plugin_mpi_comm()	comin.parallel_get_plugin_mpi_comm
comin_parallel_get_host_mpi_comm	int comin_parallel_get_host_mpi_comm()	comin.parallel_get_host_mpi_comm
comin_parallel_get_host_mpi_rank	int comin_parallel_get_host_mpi_rank()	comin.parallel_get_host_mpi_rank
comin_plugin_finish	void comin_plugin_finish(const char*,const char*)	comin.finish
'n_var_request_add	void comin_var_request_add(struct t_comin_var_descriptor,_Bool,int*)	comin.var_request_add
get	void* comin_var_get(int,int*,struct t_comin_var_descriptor,int)	comin.var_get

Source:
ComIn technical design document
<https://icon-comin.gitlab-pages.dkrz.de/comin>

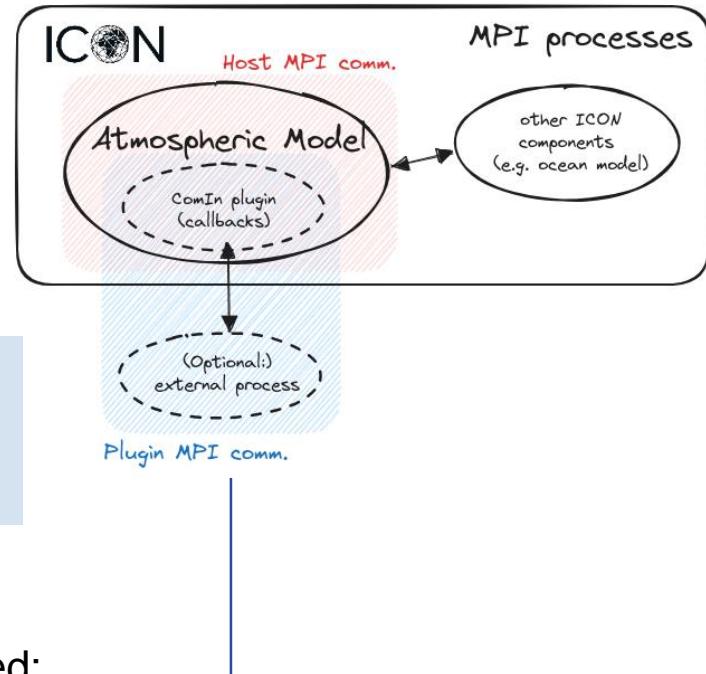


MPI communicator concept

Only process-local partitions can be accessed directly – but MPI exchanges are possible.

Plugins may handle their own parallel communication with the help of a **host communicator**.

```
comm = MPI.Comm.f2py(comin.parallel_get_host_mpi_comm())
rank = comm.Get_rank()
```



In addition, a **plugin communicator** (optional) can be created:
Useful for exchanging data with other running processes.

Limitations / Discussion

- ICON calls ComIn, not vice versa!
 - ~ plugins do not influence ICON's restart behavior
 - ~ processes in the host model are not switched off, but can only be deactivated using ICON namelist switches
- Granularity: above block loop level, only global variables are exposed
- access to the process-local MPI partition only (but MPI exchanges possible)
- no asynchronicity between ICON and the 3rd party modules

ComIn is currently under construction. Feedback welcome!





Applications



Example plugins

ComIn comes bundled together with several example plugins.

Script name	Language	Description
simple_fortran	Fortran	Simple ComIn plugin written in the <i>Fortran</i> programming language.
calc_water_column	Fortran	Simple diagnostic application, calculating the liquid water path, the ice water path, and the total water column.
yaxt_fortran	Fortran	Example plugin to demonstrate the usage of the YAXT communication library.
simple_c	C/C++	Simple ComIn plugin written in the <i>C</i> programming language.
yac_input	C/C++	Plugin written in C which encapsulates a YAC coupling inside a ComIn plugin.
yaxt_c	C/C++	Plugin written in C to demonstrate using the YAXT communication library.
simple_python_plugin.py	Python	Simple ComIn plugin written in Python, using the ComIn Python adapter.
point_source.py	Python	Test plugin requesting a tracer that participates in ICON's turbulence and convection scheme.

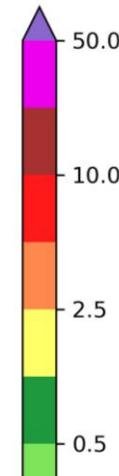


Btw.: See the practical exercises for an example how to use [matplotlib](#) directly within ICON.



Point source example

Point Source 141.0E, 37.4N, lev=10 (17km) - 2014-06-03 00 UTC



ComIn example:
python_adapter/
examples/
point_source.py

- request a tracer that participates in ICON's turbulence scheme
- add point source emissions to this tracer
- update the tracer with tendencies received from ICON's turbulence scheme
- exploit Python's vast library of useful modules (here: `scipy.spatial.KDTree`)



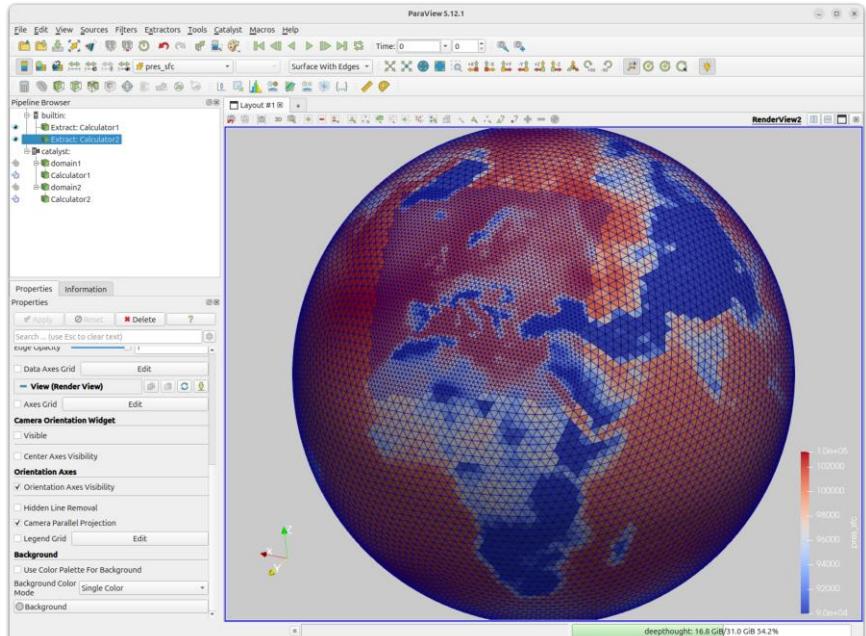
Catalyst in situ visualization



Paraview^[1] is an open-source data analysis and visualization application.

Simulation data can be streamed into Paraview using the Catalyst API specification.

- application example: Implement the Catalyst streaming for ICON as a ComIn plugin.



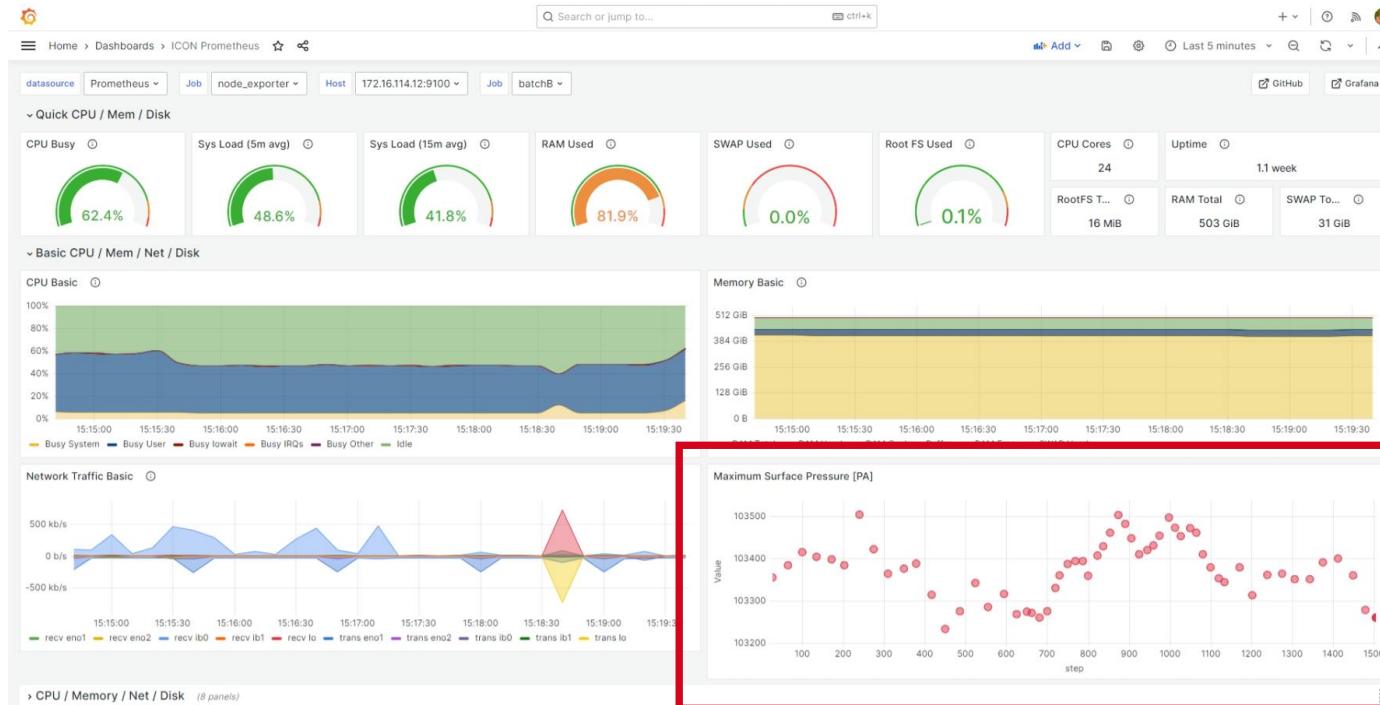
Source: N. Dreier (DKRZ)



[1] <https://gitlab.kitware.com/paraview>

Prometheus/Grafana example

Flexible monitoring dashboard offered by Grafana:



Proof-of-concept:
connect Grafana to
ICON with a ComIn
exporter plugin

Ex.: DWD NEC platform
displaying ICON's
maximum surface pressure

Source:
C. Eser (DWD-TI), FP



Wrap-up



Roadmap for ComIn v0.2.0

Features that will become available in ComIn 0.2.0

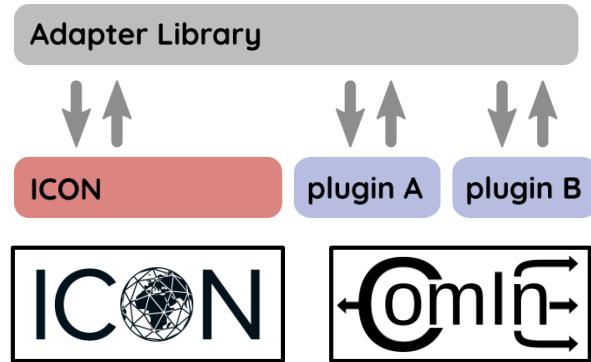
~ August 2024:

- development tool `comin_replay`
test ComIn plugins with previously recorded data sets
- GPU host-device transfer
- combining ComIn with YAC^[1]
ComIn is an interface, not a translation layer
- generalizations: additional meta-data; edge-based and vertex-based fields, ...



Contact

- GitLab-Project: <https://gitlab.dkrz.de/icon-comin/comin>
 - code (including examples and tests)
 - raise an issue
 - contribute (open a merge request)
- Extensive online documentation:
 - <https://icon-comin.gitlab-pages.dkrz.de/comin/>
 - see also the ICON Tutorial 2024 [1],
ComIn: Section 9.5
 - GMD publication (submitted)
- reach out to us: comin@icon-model.org



[1] [https://www.dwd.de/EN/ourservices/nwp_icon_tutorial/
pdf_volume/icon_tutorial2024_en.html](https://www.dwd.de/EN/ourservices/nwp_icon_tutorial/pdf_volume/icon_tutorial2024_en.html)



Florian Prill

Met. Analyse und Modellierung

Deutscher Wetterdienst

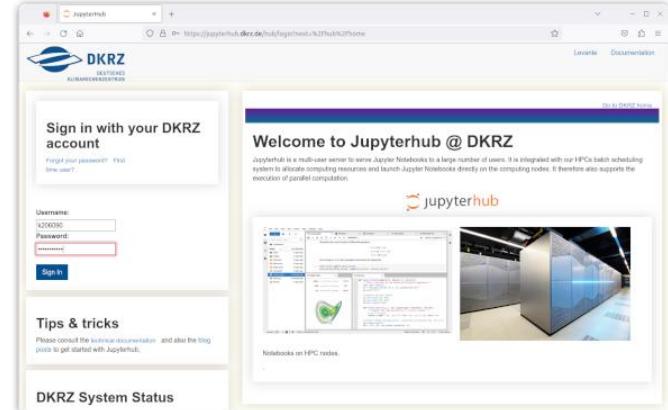
e-mail: Florian.Prill@dwd.de

ICON ComIn project team: N.-A. Dreier (DKRZ); J. Geisbüsch (DWD);
M. Haghhighatnasab (DWD); K. Hartung (DLR); P. Jöckel (DLR); A. Kerkweg
(FZJ); B. Kern (DLR); W. Loch (DKRZ); F. Prill (DWD); D. Rieger (DWD)

Practical exercises

The exercises are organized as Jupyter notebooks.

- Web site <https://jupyterhub.dkrz.de>:
Enter your username and the corresponding password
on the start page of the JupyterLab portal.



The screenshot shows two windows side-by-side. The left window is a browser showing the Jupyterhub login page for DKRZ. It has fields for 'Username' (set to 'ICON000') and 'Password', with a 'Sign in' button. Below the form is a 'Tips & tricks' section and a 'DKRZ System Status' box. The right window is a dashboard titled 'Welcome to Jupyterhub @ DKRZ'. It includes a brief description of Jupyterhub, a 'jupyterhub' logo, and two images: one of a Jupyter Notebook interface and another of a server rack labeled 'Notebooks on HPC nodes'.

- In your home directory: Extract course material from
with the following command:

```
cd $HOME  
tar xfz /pool/data/ICON/ICON_training/comin_exercises.tar.gz
```

- We will be working in the directory
\$HOME/comin-training-exercises

