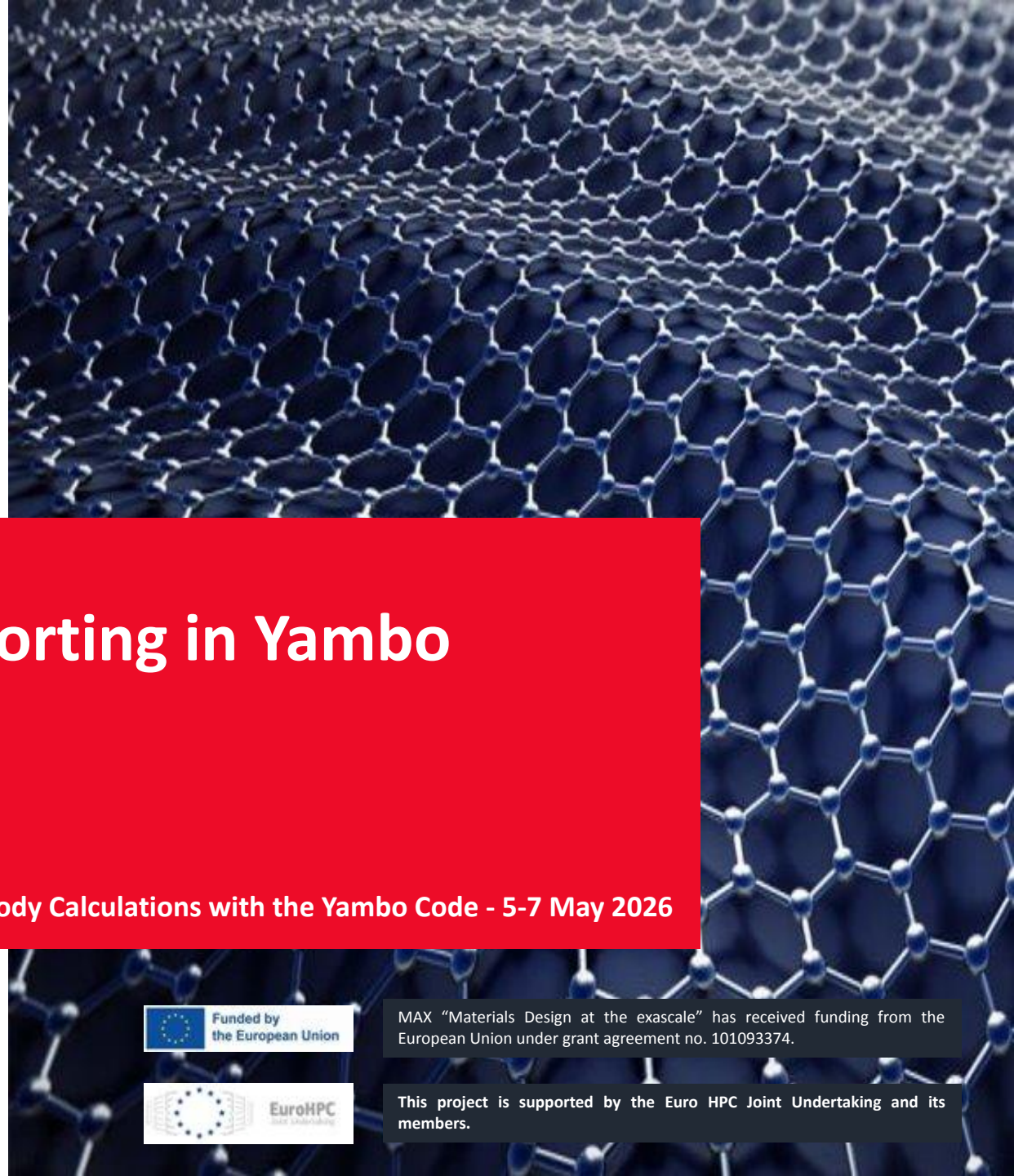
MAX DRIVING THE EXASCALE TRANSITION



Parallelism Strategy and GPU Porting in Yambo

Nicola Spallanzani
S3 Centre, Istituto Nanoscienze CNR, Modena - Italy

Computational Spectroscopy: Hands-on Many-Body Calculations with the Yambo Code - 5-7 May 2026

Funded by the European Union

MAX "Materials Design at the exascale" has received funding from the European Union under grant agreement no. 101093374.

EuroHPC

This project is supported by the Euro HPC Joint Undertaking and its members.

MaX: Goal

First exascale supercomputer:

Frontier (@ORNL)

- 1100 PFlops
- 37888 GPUs (AMD MI250X)

We need to turn MaX lighthouse codes into exascale-enabled applications:

- **large scale MPI parallelism** (order of 10000 tasks)
- combined with **GPU awareness**

Single-node optimisation

- make sure MaX codes can exploit accelerated nodes featuring multiple GPU brands

Multi-node parallel efficiency

- make the codes scalable in the presence of GPUs

Scientific software engineering

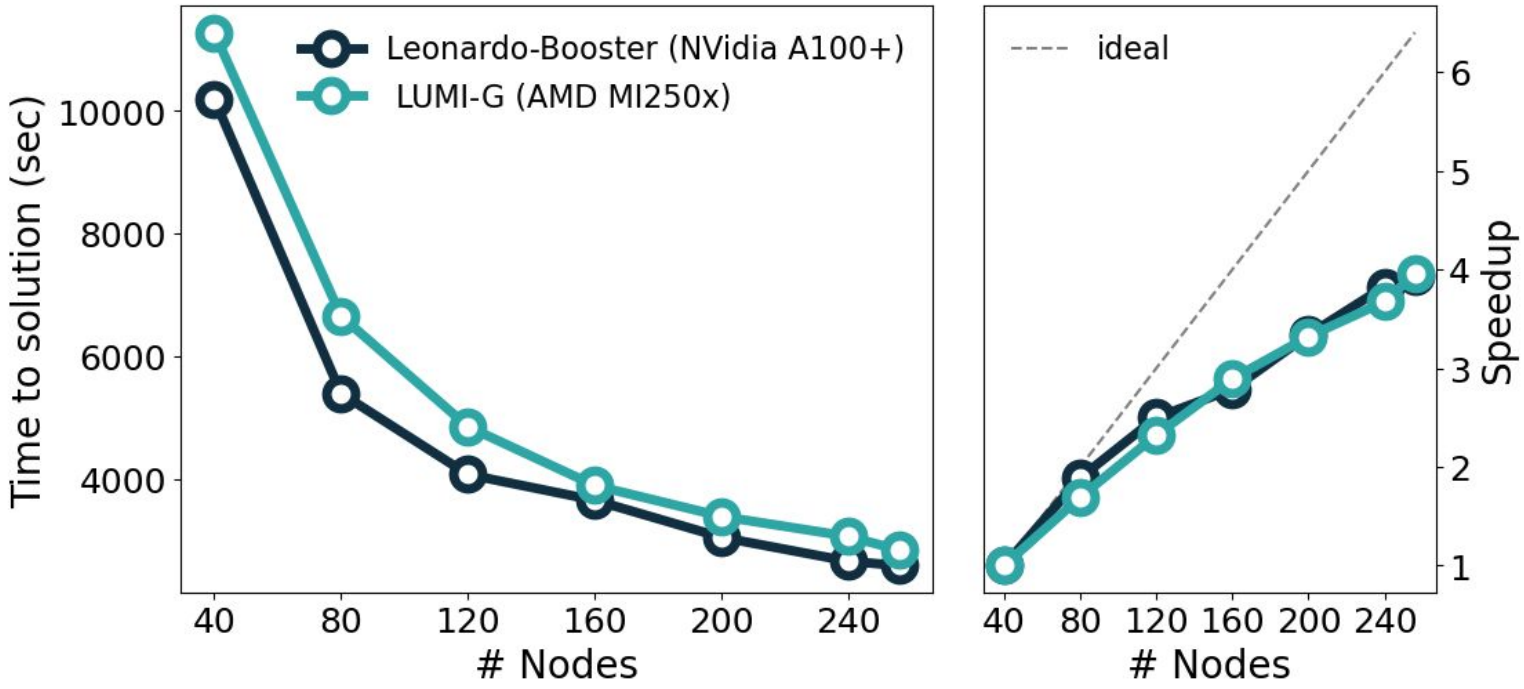
- support long-term maintainability and community contributions

New Scientific features

YAMBO: on pre-exascale

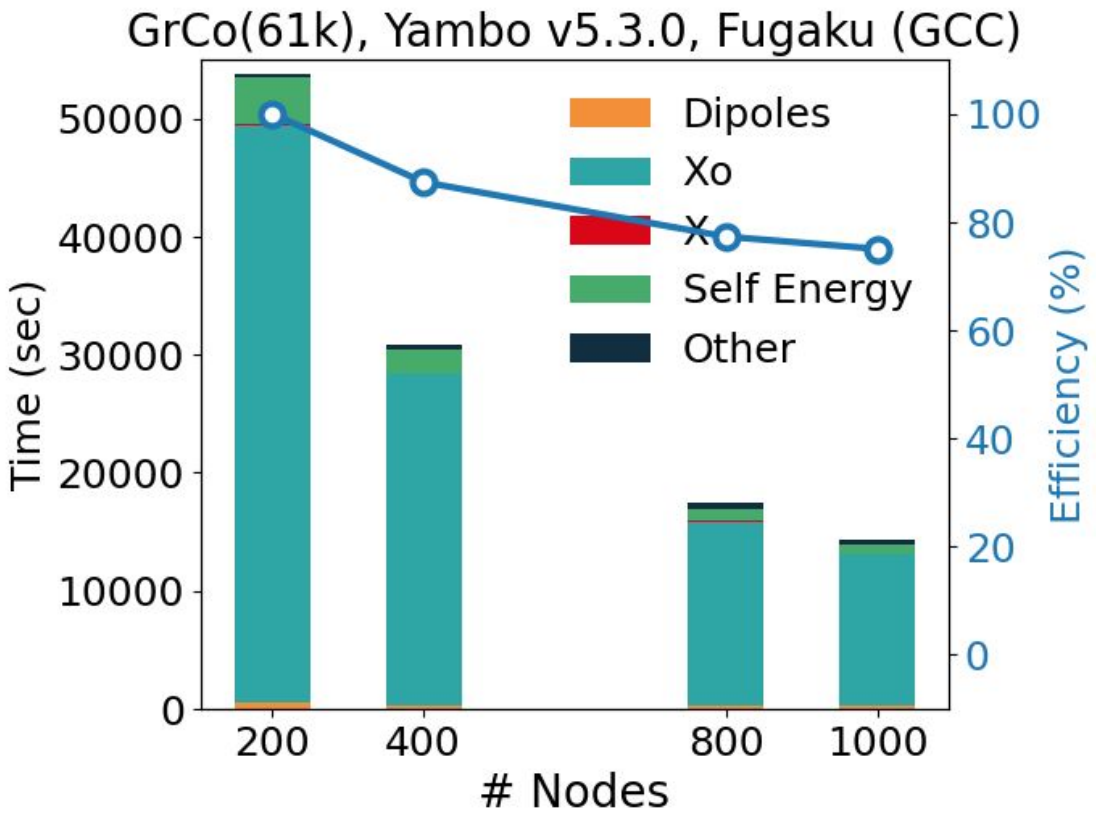
- Currently running on **NVIDIA, AMD, and INTEL accelerated machines**
- Using **CUDA-Fortran, OpenACC, and OpenMP** programming models
- Integration of deviceXlib, a MaX component for wrapping device-oriented routines and utilities

Strong scaling GW workflow, architecture x86_64 + GPU - Yambo v5.3.0



YAMBO: on pre-exascale

- Running on ARM architectures like Supercomputer Fugaku



Benchmarks of MaX applications: https://gitlab.com/max-centre/benchmarks-max3

The YAMBO code: Linear Response

$$\chi(\mathbf{q},\omega) = [I - \chi_0(\mathbf{q},\omega)v(\mathbf{q})]^{-1}\,\chi_0(\mathbf{q},\omega)$$

$$\chi^0_{\mathbf{G}\mathbf{G}'}(\mathbf{q},\omega) = 2\sum_{c,v}\int_{BZ}\frac{d\mathbf{k}}{(2\pi)^3}\,\rho^*_{cv\mathbf{k}}(\mathbf{q},\mathbf{G})\rho_{cv\mathbf{k}}(\mathbf{q},\mathbf{G}')\,f_{v\mathbf{k}-\mathbf{q}}(1-f_{c\mathbf{k}})\times$$
$$\times\left[\frac{1}{\omega+\epsilon_{v\mathbf{k}-\mathbf{q}}-\epsilon_{c\mathbf{k}}+i0^+}-\frac{1}{\omega+\epsilon_{c\mathbf{k}}-\epsilon_{v\mathbf{k}-\mathbf{q}}-i0^+}\right]$$

q transferred momenta (MPI q)

Xo bands (MPI c,v)

k momenta (MPI k)

Space variables (MPI g)

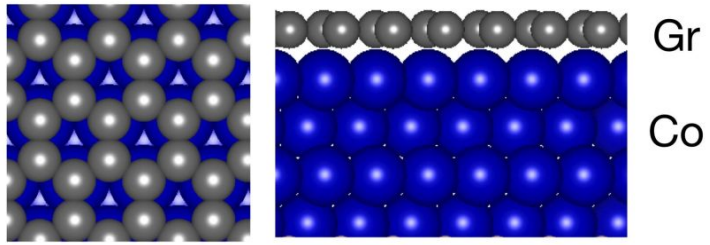
```
X_and_IO_CPU= "1 1 2 16 2"
X_and_IO_ROLEs= "q g k c v"
X_and_IO_nCPU_LinAlg_INV= 64
X_Threads= 4
```

- MPI-c,v best memory distribution
- MPI-k as efficient, some memory duplication
- MPI-q may leads to load unbalance and mem duplication
- OpenMP efficient, need extra memory

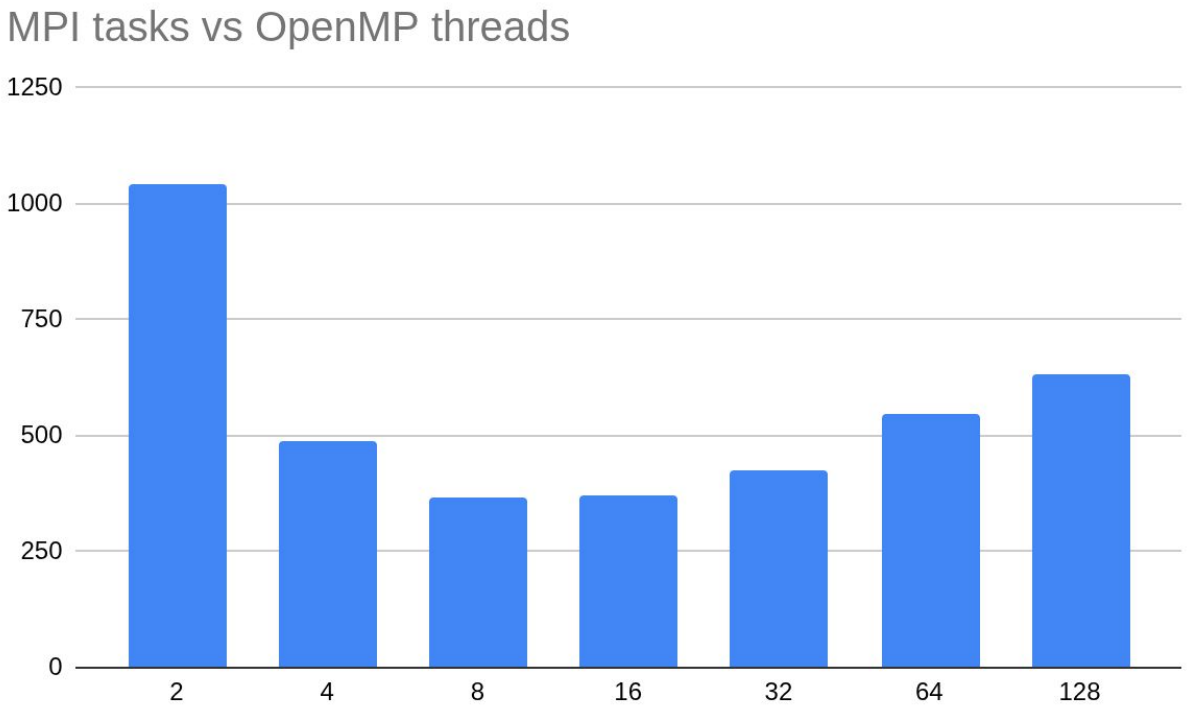
The YAMBO code: MPI tasks vs. OpenMP threads

GrCo benchmark:

- complete GW workflow for a graphene/Co interface (GrCo) composed by a graphene sheet adsorbed on a Co slab 4 layers thick, and a vacuum layer as large as the Co slab



Nodes	MPI tasks	Tasks / Node	OMP threads	Time
2	4	2	64	1043.9282
2	8	4	32	490.3788
2	16	**8**	**16**	**364.8921**
2	32	**16**	**8**	**369.5022**
2	64	32	4	425.2116
2	128	64	2	547.0049
2	256	128	1	633.9409



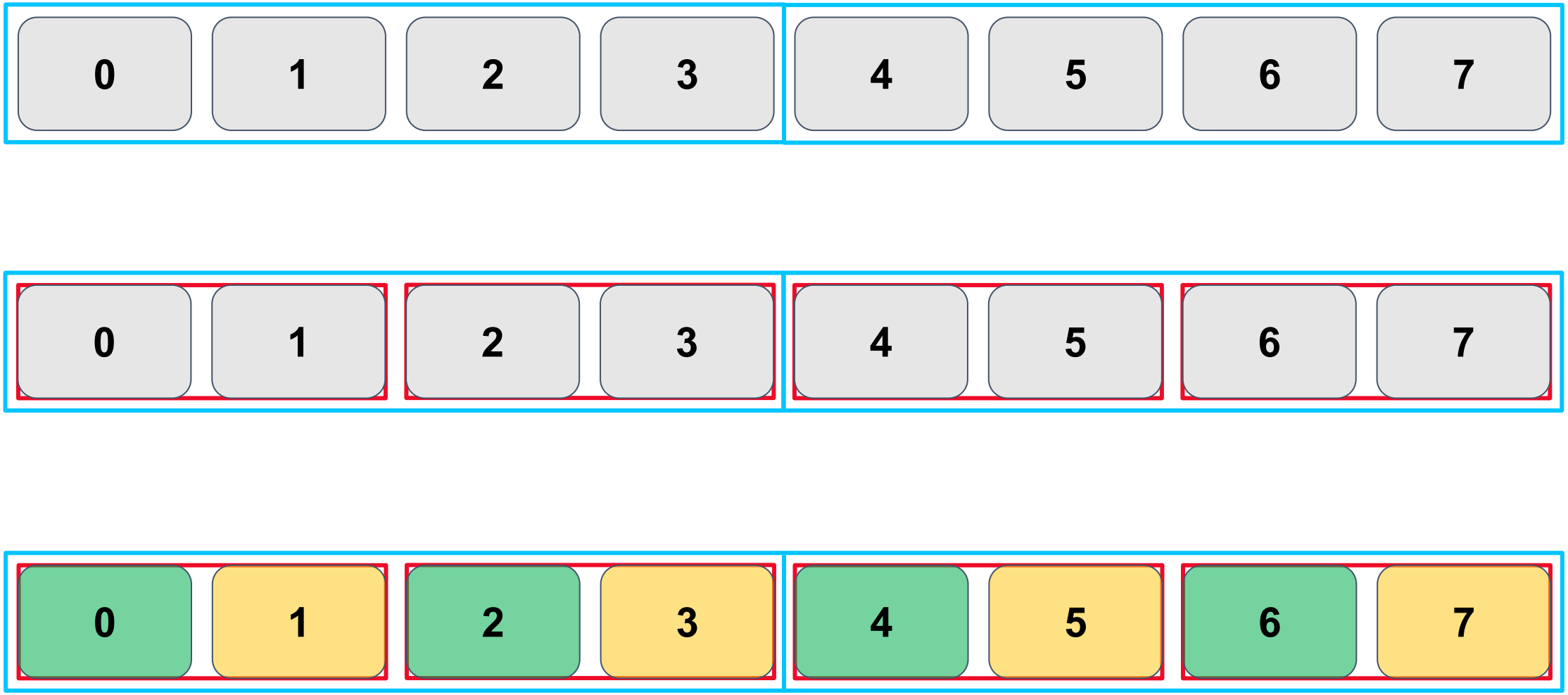
The YAMBO code: Linear Response, Parallelization Tests

MPI Tasks	Q	G	K	C	V	Time
16	1	1	1	16	1	362.9453
16	1	1	1	8	2	420.2962
16	1	1	1	4	4	418.7133
16	1	1	1	2	8	453.233
16	1	1	2	8	1	426.182
16	1	1	4	4	1	504.3008
16	1	2	1	8	1	394.941
16	1	4	1	4	1	500.1078
16	1	8	1	2	1	808.5955
16	**2**	**1**	**1**	**8**	**1**	**362.8131**
16	4	1	1	4	1	410.3318
16	2	2	2	2	1	384.5513
16	1	16	1	1	1	1438.3895
16	4	1	4	1	1	541.0781

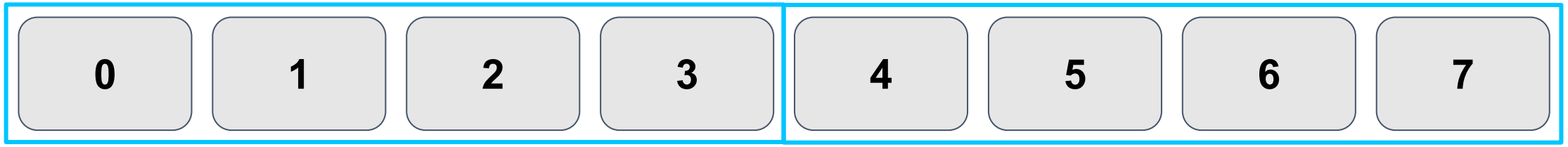
MPI Tasks	Q	G	K	C	V	Time
32	2	2	2	2	2	388.1418
32	2	2	2	4	1	370.668
32	1	2	2	8	1	393.4069
32	2	1	2	8	1	453.1927
32	**2**	**2**	**1**	**8**	**1**	**328.6743**
32	2	1	1	16	1	365.434
32	1	2	1	16	1	332.6477
32	1	1	2	16	1	432.7082
32	1	1	1	32	1	367.4084
32	1	4	1	8	1	471.9442
32	1	8	1	4	1	770.5978
32	1	1	4	8	1	506.2065
32	4	1	1	8	1	427.9842
32	1	16	1	2	1	1382.2635

Q	G	K	C	V
7	**675**	**7**	**960**	**40**

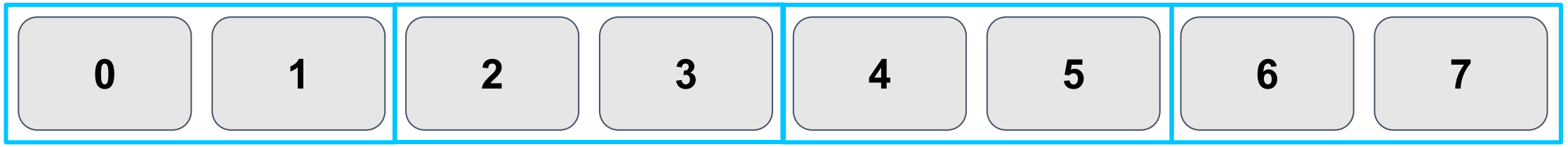
The YAMBO code: MPI parallelization hierarchy



The YAMBO code: workload parallel distribution



Computing: 4 q-points 3 q-points



Computing: 2 q-points 2 q-points 2 q-points 1 q-point

GPU-aware programming style

Within the **MaX CoE** framework, **YAMBO** has undergone a significant source code refactoring effort in recent years, primarily aimed at improving:

1. Readability
2. Maintainability

with particular emphasis on the removal of GPU-related code duplication.

The adopted solutions include:

- The use of wrappers (e.g., deviceXlib library)
- The use of preprocessor macros
- The integration of GPU-aware external libraries (e.g., linear algebra libraries, FFT libraries)

What is deviceXlib?

- deviceXlib is a library that wraps device-oriented routines and utilities, such as device data allocation, host-device data transfers.
- deviceXlib supports CUDA language, together with OpenACC and OpenMP programming paradigms.
- deviceXlib wraps a subset of functions from Nvidia cuBLAS, Intel oneMKL BLAS and AMD rocBLAS libraries.
- currently working with
  - NVIDIA + CUDA-Fortran
  - NVIDIA + OpenACC
  - AMD + OpenMP-GPU
  - INTEL + OpenMP-GPU

Recent developments

- Repository: https://gitlab.com/max-centre/components/devicexlib
- Recently completely restructured (with the help of G. Rossi, INTEL)
- Tests suite added and bugs fixed
- DeviceXlib version 0.9.1 tagged
- Fully tested on Leonardo and LUMI (among others)

deviceXlib: Structure

```
module devXlib

  use devxlib_malloc
  use devxlib_memcpy
  use devxlib_mapping
  use devxlib_buffers
  use devxlib_linalg
  use devxlib_auxfunc
  use devxlib_async

  implicit none
end module devXlib
```

- devxlib_alloc
- devxlib_free
- devxlib_allocated

- devxlib_memcpy_h2h
- devxlib_memcpy_d2d
- devxlib_memcpy_d2h
- devxlib_memcpy_h2d
- devxlib_memcpy_d2d_p
- devxlib_memcpy_d2h_p
- devxlib_memcpy_h2d_p

- devxlib_map
- devxlib_unmap
- devxlib_mapped

- devxlib_xDOT
- devxlib_xDOT_gpu
- devxlib_xGEMM
- devxlib_xGEMM_gpu

- devxlib_conjg
- devxlib_vec_upd_remap
- devxlib_mat_upd_dMd
- devxlib_mem_addscal

deviceXlib: Pre-compiler Macros

- Pre-compiler macros (DEV_VAR, DEV_SUB, DEV_ATTR) to hide differences across cpu and gpu code, e.g. appending _d or _gpu to variable and subroutine names.

```
#ifdef __DXL_CUDAF
#    define DEV_SUB(x) x##_gpu
#    define DEV_VAR(x) x##_d
#    define DEV_ATTR , device
#elif defined __DXL_OPENACC || defined __DXL_OPENMP_GPU
#    define DEV_SUB(x) x##_gpu
#    define DEV_VAR(x) x
#    define DEV_ATTR
#else
#    define DEV_SUB(x) x
#    define DEV_VAR(x) x
#    define DEV_ATTR
```

include/devxlib_defs.h

deviceXlib: Pre-compiler Macros

```
#if defined __DXL_CUDAF    ||
    defined __DXL_OPENACC  ||
    defined __DXL_OPENMP_GPU
#  define __DXL_HAVE_DEVICE
#endif

/*
! directive sentinels
*/

#if defined __DXL_CUDAF
#  define DEV_CUF $cuf
#else
#  define DEV_CUF !!!!
#endif
```

include/devxlib_macros.h

```
#if defined __DXL_OPENACC
#  define DEV_ACC $acc
#else
#  define DEV_ACC !!!!
#endif


#if defined __DXL_OPENMP_GPU
#  define DEV_OMPGPU $omp
#else
#  define DEV_OMPGPU !!!!
#endif

#if defined __DXL_OPENMP && !defined
(__DXL_HAVE_DEVICE)
#  define DEV_OMP $omp
#else
#  define DEV_OMP !!!!
#endif
```

YAMBO: deviceXlib Wrappers

```
use devxlib, ONLY:devxlib_memcpy_d2d
[...]
do jb=Sx_lower_band,Sx_upper_band
    [...]
    call DEV_SUB(scatter_Bamp)(isc)
    [...]
    if (isc%is(1)/=iscp%is(1)) then
        call DEV_SUB(scatter_Bamp)(iscp)
    else
        ! dev2dev, iscp%rhotw = isc%rhotw
        call devxlib_memcpy_d2d(DEV_VAR(iscp%rhotw),DEV_VAR(isc%rhotw))
    endif

    DP_Sx_l=DEV_SUB(Vstar_dot_VV)(isc%ngrho,DEV_VAR(iscp%rhotw),&
            DEV_VAR(isc%rhotw),DEV_VAR(isc%gamp)(:,1))
    DP_Sx=DP_Sx + DP_Sx_l * const
enddo
```

Macros:
DEV_SUB: append "_gpu" to the name of the subroutine
DEV_VAR: append "_d" to the name of the variable
DEV_ATTR: substitute ", device" when CUDA-Fortran is enabled

YAMBO: single source code

- Single source code
- Simple Fortran data structures to organize cpu and gpu variables

```
subroutine DEV_SUB(scatter_Bamp)(isc)
[...]
complex(SP), pointer DEV_ATTR :: WF_symm_i_p(:,:), WF_symm_o_p(:,:)
complex(SP), pointer DEV_ATTR :: rhotw_p(:)
complex(DP), pointer DEV_ATTR :: rho_tw_rs_p(:)
!
! define pointers to enable CUF kernels
! when compiling using CUDA-Fortran
!
WF_symm_i_p => DEV_VAR(isc%WF_symm_i)
WF_symm_o_p => DEV_VAR(isc%WF_symm_o)
rho_tw_rs_p => DEV_VAR(isc%rho_tw_rs)
rhotw_p => DEV_VAR(isc%rhotw)
[...]
```

Macros:
DEV_SUB: append “_gpu” to the name of the subroutine
DEV_VAR: append “_d” to the name of the variable
DEV_ATTR: substitute “, device” when CUDA-Fortran is enabled

YAMBO: Concurrency of Backends

Macros:
DEV_CUF: substitute “$cuf” when CUDA-Fortran is enabled
DEV_ACC: substitute “$acc” when OpenACC is enabled
DEV_ACC_DEBUG: substitute “$acc” when OpenACC and debug are enabled
DEV_OMPGPU: substitute “$omp” when OpenMP-GPU is enabled
DEV_OMP: substitute “$omp” when OpenMP is enabled

```
complex(SP), pointer DEV_ATTR ::
           WF_symm_i_p(:,:),
           WF_symm_o_p(:,:)
complex(DP), pointer DEV_ATTR ::
           rho_tw_rs_p(:)

WF_symm_i_p =>
           DEV_VAR(isc%WF_symm_i)
WF_symm_o_p =>
           DEV_VAR(isc%WF_symm_o)
rho_tw_rs_p =>
           DEV_VAR(isc%rho_tw_rs)
```

```
!DEV_ACC_DEBUG data present(rho_tw_rs_p,WF_symm_i_p,WF_symm_o_p)
!DEV_ACC parallel loop async
!DEV_CUF kernel do(1) <<<*,*>>>
!DEV_OMPGPU target map(present,alloc:rho_tw_rs_p,WF_symm_i_p,WF_symm_o_p)
!DEV_OMPGPU teams loop
!DEV_OMP parallel default(shared), private(ir)
!DEV_OMP do
do ir = 1, fft_size
  rho_tw_rs_p(ir) = cmplx(conjg(WF_symm_i_p(ir,1))*WF_symm_o_p(ir,1),kind=DP)
enddo
!DEV_OMPGPU end target
!
if (n_spinor==2) then
  !DEV_ACC parallel loop async
  !DEV_CUF kernel do(1) <<<*,*>>>
  !DEV_OMPGPU target map(present,alloc:rho_tw_rs_p,WF_symm_i_p,WF_symm_o_p)
  !DEV_OMPGPU teams loop
  !DEV_OMP do
  do ir = 1, fft_size
    rho_tw_rs_p(ir) = &
        rho_tw_rs_p(ir)+cmplx(conjg(WF_symm_i_p(ir,2))*WF_symm_o_p(ir,2),kind=DP)
  enddo
  !DEV_OMPGPU end target
endif
!DEV_OMP end parallel
!DEV_ACC_DEBUG end data
```

deviceXlib

	CUDA Fortran	OpenACC	OpenMP

Arrays allocation	`allocate`
	`deallocate`
	`allocated`
	using `device` attribute
		`enter data create`
		`exit data delete`
		`acc_is_present`
		on host memory
			`enter data map`
			`exit data map`
			`omp_target_is_present`
			on host memory

Pointers allocation (GPU only)	`cudaMalloc`
	`cudaFree`
	`associated`
		`acc_malloc`
		`acc_free`
		`associated`
			`omp_target_alloc`
			`omp_target_free`
			`associated`

Memory copies	**Arrays:** direct copy
	Pointers: `cudaMemcpy`
		Arrays: `update device`,
		`update host`,
		`acc parallel loop`
		Pointers:
		`acc_memcpy_to_device`,
		`acc_memcpy_from_device`
		`acc_memcpy_device`
			Arrays: `update to`,
			`update from`,
			`omp target teams loop`

Linear algebra	cuBLAS	cuBLAS	oneMKL rocBLAS

Thank you for your attention!

MaX DRIVING THE EXASCALE TRANSITION

JOIN THE COMMUNITY NOW!

Follow us on:

company/max-centre/

http://www.max-centre.eu/

@max_center2

youtube/channel/MaX Centre eXascale

Funded by the European Union

MaX "*Materials Design at the Exascale*" has received funding from the European Union under grant agreement no. 101093374.

EuroHPC Joint Undertaking

The project is supported by the Euro HPC Joint Undertaking and its members.