Part 2 – High-Performance Computing – NEMO Climate Simulation Application

Strategy

- NEMO
  NEMO which stands for Nucleus for European Modelling of the Ocean, is a state-of-the-art modelling framework for research activities and forecasting services in ocean and climate sciences, developed in a sustainable way by a European consortium. (https://www.nemo-ocean.eu/) NEMO is distributed with several reference configurations, allowing both the user to set up a first application and the developer to validate the Developments. (Figure 2.2)

- GYRE spec
  The GYRE configuration has been built to simulate the seasonal cycle of a double-gyre box model. The domain geometry is a closed rectangular basin on the beta-plane centred at sin(30) and rotated by 45°, 3180 km long, 2120 km wide and 4 km deep. (Figure 2.3) The domain is bounded by vertical walls and by a flat bottom. The configuration is meant to represent an idealized North Atlantic or North Pacific basin. The circulation is forced by analytical profiles of wind and buoyancy fluxes.

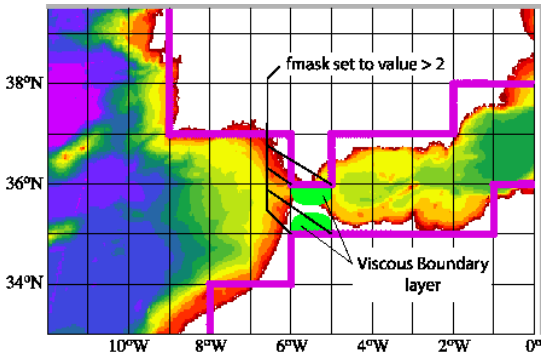


Figure 2.1

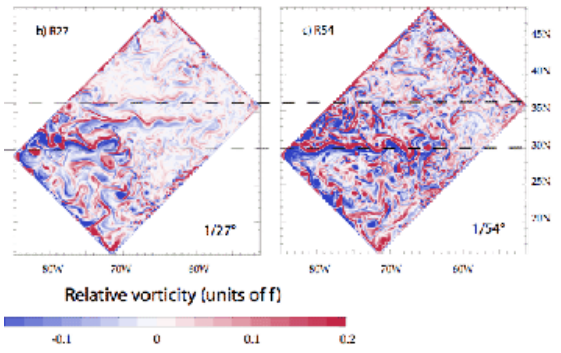


Figure 2.2

- Profile analysis

  We did profile analysis to understand the performance of MEMO. As the diagram shows, the function "_traadv_fct_MOD_tra_adv_fct" is the hotspot of the NEMO model program. It consumes 32.8% computing resources of the total computing sum. Through the profile diagram (Figure 2.3), we learned more about the NEMO computing resource consuming distribution. The information also helps us to determine which part of the code could be ported to GPU for acceleration. Although we can't use GPUs in this competition, we did find some references and documentations, and we would very much like to try it in the future.

Function Stack	CPU Time: Total ▼	CPU Time: Self	Instructions Retired: Total	Instructions Retired: Self	Microarchitecture Usa
Total	750.130s	0s	100.0%	0	
__traadv_fct_MOD_tra_adv_fct	171.215s	171.215s	32.8%	1,082,440,000,000	
__traldf_iso_MOD_tra_ldf_iso	117.405s	117.405s	15.1%	500,310,000,000	
__ieee754_exp_avx	61.495s	61.495s	11.4%	376,180,000,000	
clear_page_c_e	52.665s	52.665s	0.0%	1,250,000,000	
__ldfslp_MOD_ldf_slp	26.645s	26.645s	4.0%	131,480,000,000	
__dynspg_ts_MOD_dyn_spg_ts	21.575s	21.575s	2.0%	67,560,000,000	
__trazdf_MOD_tra_zdf_imp	20.180s	20.180s	1.7%	57,490,000,000	
__zdftke_MOD_zdf_tke	19.140s	19.140s	2.3%	77,480,000,000	
__p2zbio_MOD_p2z_bio	17.975s	17.975s	1.8%	60,600,000,000	
xios::CAttributeArray<double, (int)1>::s	15.640s	15.640s	2.4%	80,720,000,000	
__ieee754_log_avx	14.870s	14.870s	2.5%	81,260,000,000	
__eosbn2_MOD_rab_3d	12.410s	12.410s	1.6%	53,750,000,000	

Figure 2.3

- Installation

  We make a diagram (Figure 2.4) to show our installation process which compares within our own local cluster and NSCC machine.

  - Our own local cluster

    First, we follow the HPC-AI NEMO installation guideline and use the hpc-x mpi to build the NEMO environment on our own machine. The installation was completed smoothly throughout the guideline process. After installation success, we try to run the NEMO model using infiniband to speed up our program. Besides, we also try different mpi versions and libraries on our machine.

  - NSCC machine

    When we started installing on the NSCC machine, we encountered some errors which we couldn't solve. The first one is using hpcx mpi which leads to the compiler can't finding the c compiler. We try to install our own new cmake and solve the error. After we solved it, at

the stage of running NEMO would have a segmentation fault. We make our effort to solve the error and install repeatedly many times, the error still exists. After trying the hpcx mpi version, we installed many versions of mpi including openmpi, mpich, Openmpi from the NSCC module file, finally we ran successfully using mpich. Mpich is the only one successful version of mpi compared with OpenMPI and hpcx mpi. For the NSCC machine, we choose mpich-3.1.4 version as our mpi version.

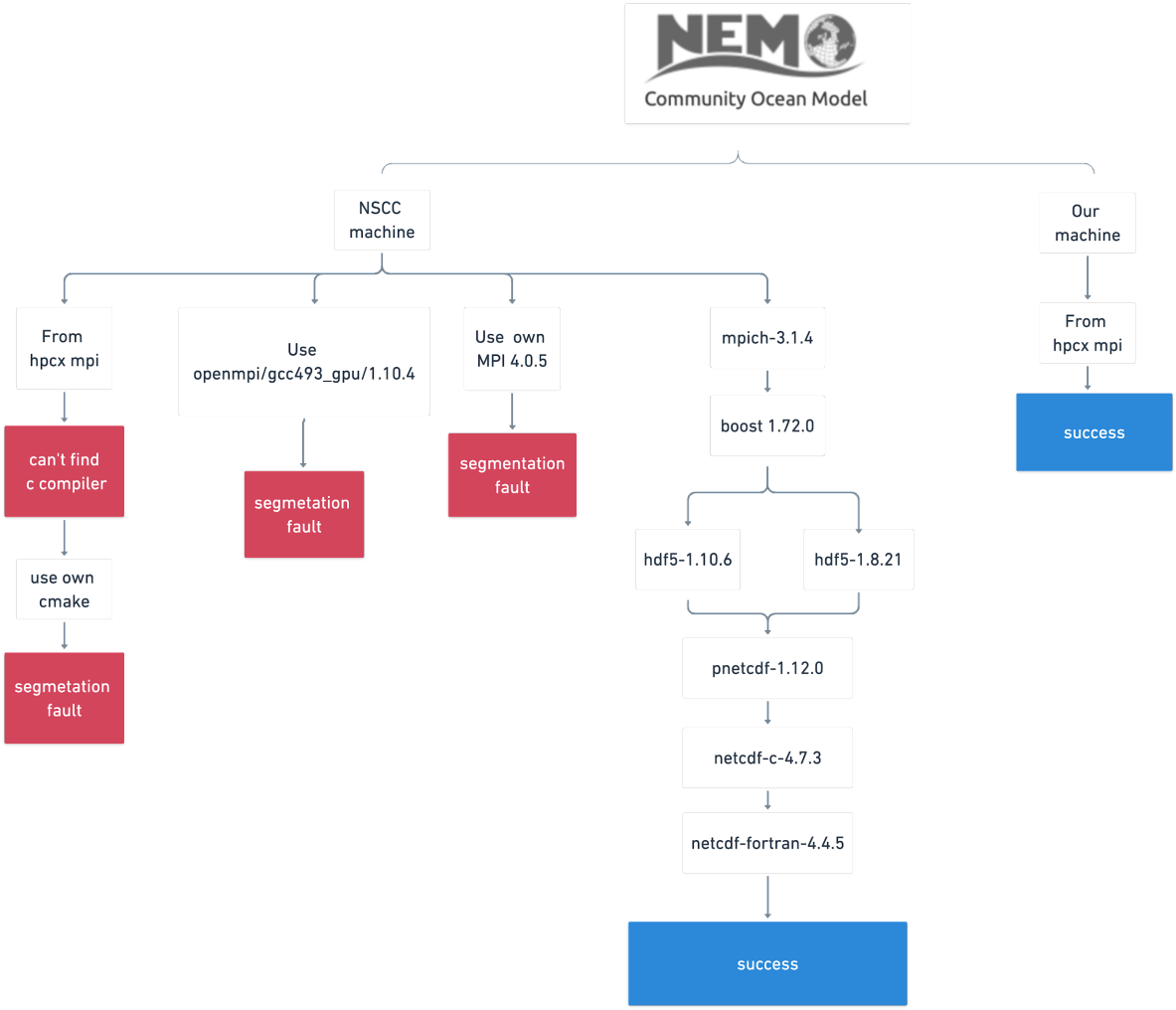


Figure 2.4

Optimizing

- Multiple nodes
  We use our script to get each hostname of node, and try to cross nodes from one node started. We get errors when running NEMO using 17 nodes and each node has 24 CPUs. The execute program always failed and couldn't be run successfully. The question confused us for a long time, we were very confused about why the program couldn't be run successfully when using more than 384 CPUs. After our team discussion, our conclusion is that the spec of the test case "nn_Gyre" has limited the number of CPUs. For the program parameter "nn_Gyre" 25 only allows 384 CPUs to run. If we want to run more than 16 nodes with 384 CPUs in total, we should distribute 384 processes equally to each node. Therefore, 384 CPUs distribute to 16 nodes and each node has 12 CPUs. After our experiment (Figure 2.5), using 32 nodes 12 CPUs per node could be run successfully and was the fastest one compared with the other versions using 384 CPUs.

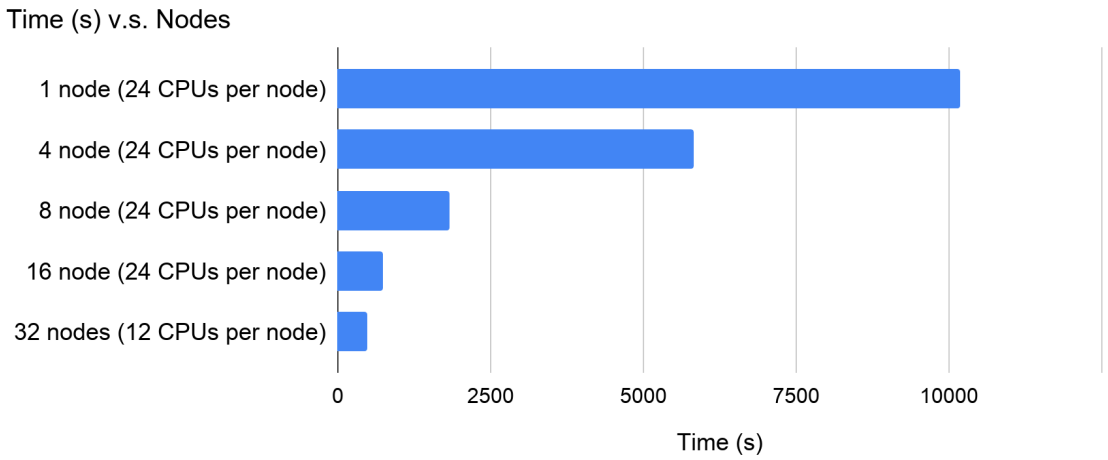


Figure 2.5

However, we received the mail from NSCC organizer, they told us that NEMO can't run with an arbitrary number of MPI ranks. After we know that, we continuously test 32 nodes with more than 12 CPUs per node. The result (Figure 2.6) shows that only four various numbers of CPUs per node could be run.

CPUs per node	Time (second)
12	475
14	460
15	446
16	fail
17	fail
18	fail

CPUs per node	Time (second)
19	fail
20	fail
21	450
22	fail
23	fail
24	fail

Figure 2.6

- -O option flag
  Compiling the makenemo file using arch-linux_gfortran.fcm file has many option flags and we discovered that %CFLAGS and %FCFLAGS there are two flags that could be fixed and may have a chance to optimize the program. The O2 O3 and Ofast compare with 16 nodes each node has 24 processes.(Figure 2.7) The testing result is that the Ofast flag is faster than the O2 flag and O3 is the slowest of them. When we test 32 nodes (12 PCUs per node) using the Ofast flag, the program would fail and can't be run. After the test, we decided to use default flag O2 to make sure that the answer and output files will be correct and fine.

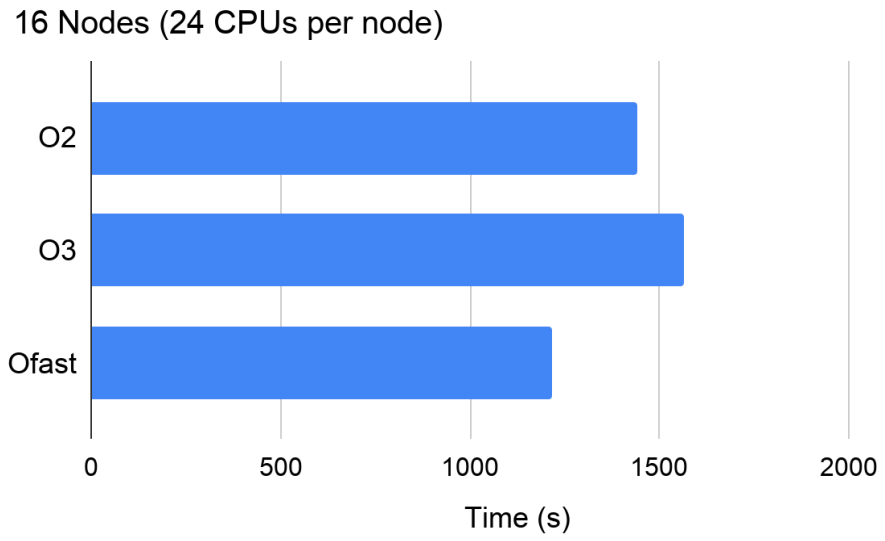


Figure 2.7

- HDF5 version
  From the hpcx NEMO guideline,the hdf5 version used in the installation tutorial is 1.10.6, but we are interested in whether different hdf5 versions could improve the performance or not. Hence, we test two different hdf5 versions on 32 nodes and each node has 12 CPUs. After multiple tests (Figure 2.8), we choose each version which has the least running time to compare. Finally, we decided to choose the hdf5 1.8.21 version, because the version is stable and a little faster than hdf5-1.10.6.

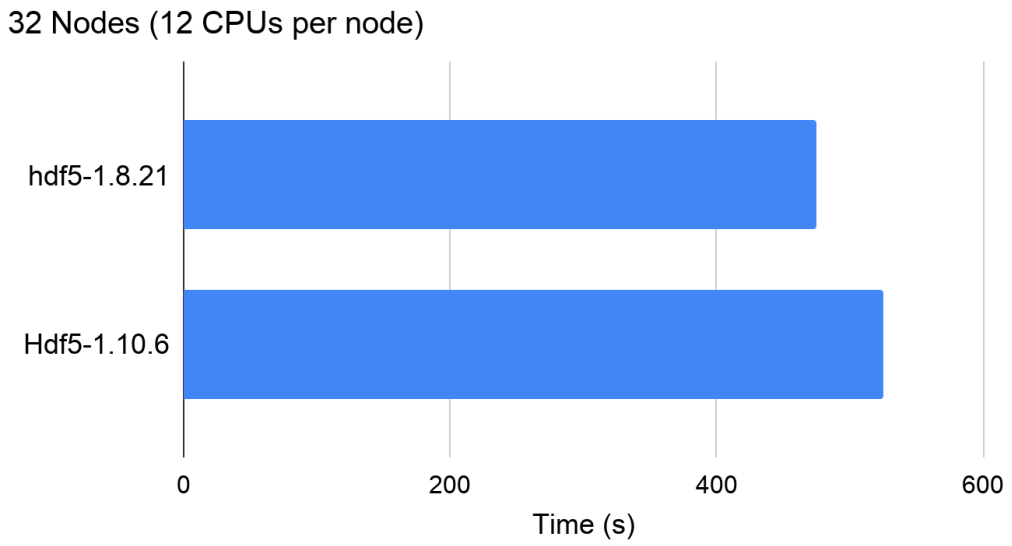


Figure 2.8

- Process-core Binding
  `-bind-to <object[:num]>` Specify the hardware element to bind each process. We make an experiment : add `-bind-to hwthread` on different numbers of nodes. Through the experiment, we discovered a special result as shown below(Figure 2.9). The result shows that using 4 nodes(24 CPUs per node) would have a significant optimizing effect. On the contrary, we used more than 16 nodes, and we got two results from using 16 nodes(24 CPUs per node) and 32 nodes (12 CPUs per node). Both of the results show that the performance would be worse than no add `-bind-to hwthread`, so we decide to remove this to optimize in our command.

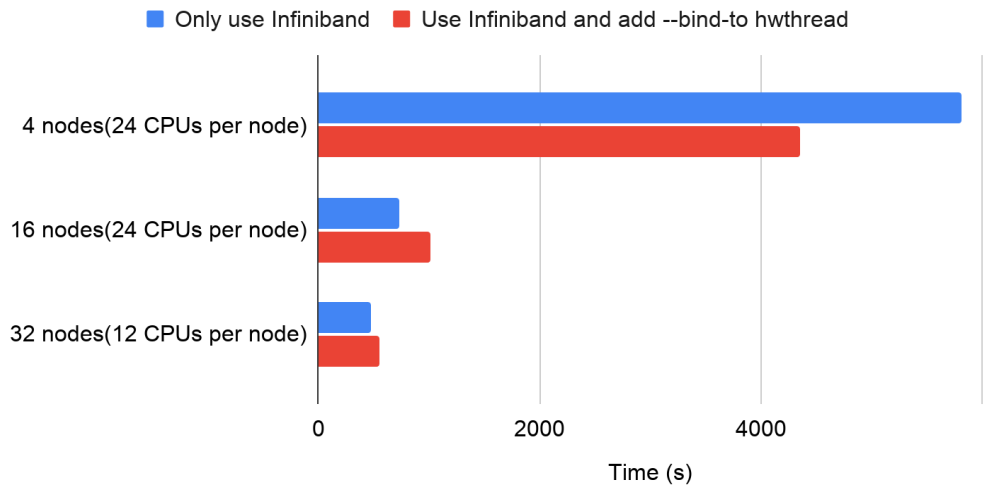


Figure 2.9

Execution

After the above testing, we finally choose the best performance version to hand out. We select the top two versions among all versions tested, and we test many times and compare their average running time to choose one of them as our best version. The best one is using 32 nodes(15 per CPUs).

> Install NEMO from GitHub or Compressed File (Choose one), and submit with the pbs file.

- Option1 : Install From GitHub

```
wget
https://raw.githubusercontent.com/William-Mou/module_file/main/instal
l_nemo.sh
chmod +x install_nemo.sh && ./install_nemo.sh
```

- Option2 : Install From Compressed File

```
tar zxvf install_nemo.tar.gz && \
cd install_nemo && \
chmod +x install_nemo_local.sh && \
./install_nemo_local.sh
```

- Evaluation GLUE benchmark

```
qsub $APPROOT/module_file/NTHU_NEMO.pbs
```

Result

- Software Compilation Version

gcc	7.5.0
boost	1.72.0
mpich	3.1.4
zlib	1.2.11
hdf5	1.8.21
pnetcdf	1.12.0
netcdf-c	4.7.3
netcdf-fortran	4.4.5
Option flag	Default

- Hardware Usage

Nodes	32
CPU per node	15
Total CPUs	480

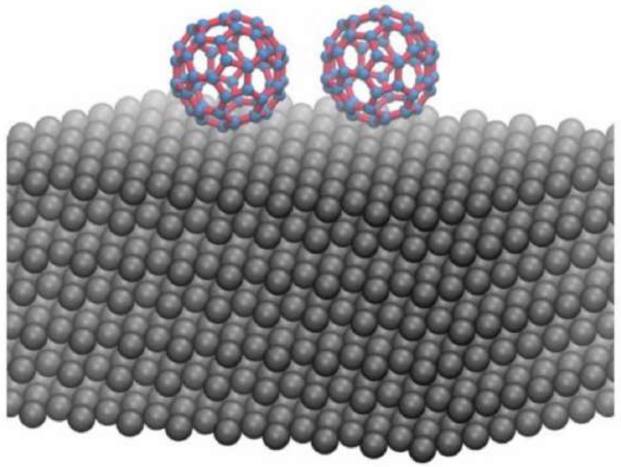
- Time

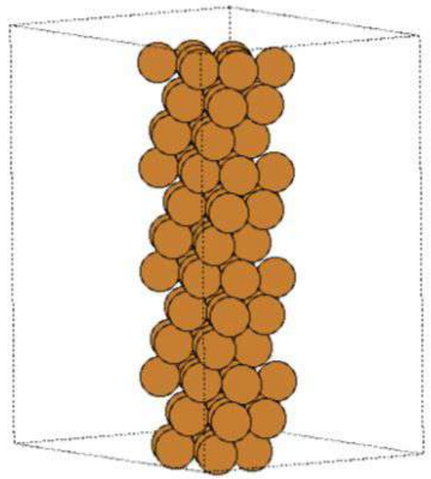
Time	00:07:26
Steps	4320
Steps/S	9.68

GPAW Introduction

- GPAW is a versatile software package for first-principles simulations of nanostructures utilizing density-functional theory and time-dependent density-functional theory.

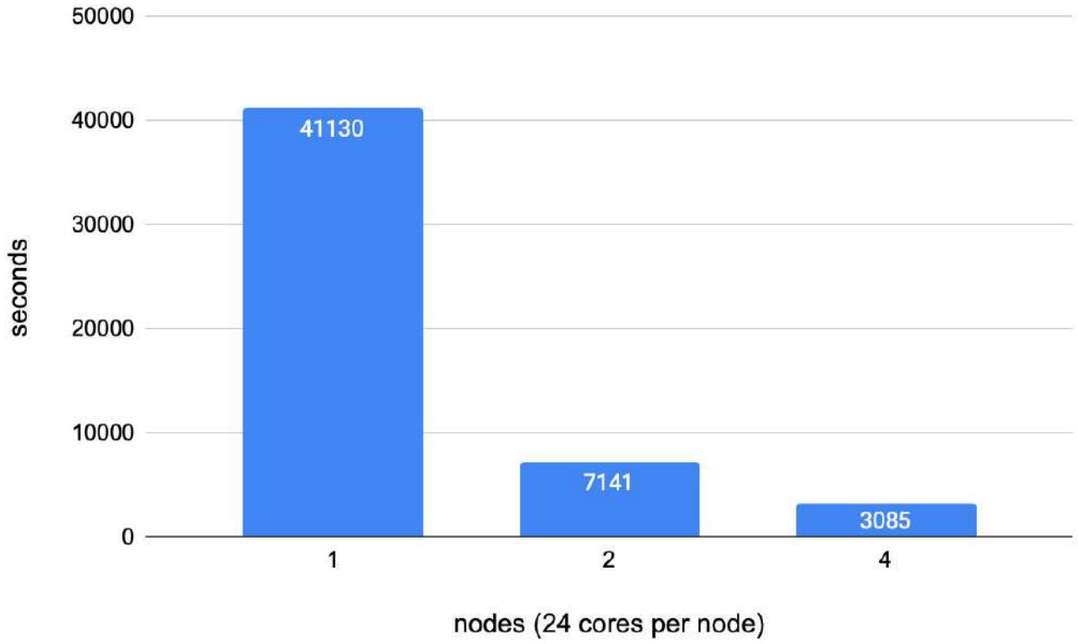
- The benchmark of GPAW in this competition is copper filament, periodic in z-direction Real-space basis, k-points in z- dimension.





NSCC using pure MPI parallelization

- **Multi-nodes has a good scalability on NSCC cluster.**
- **Using mpich to parallelize, the performance is bad.**

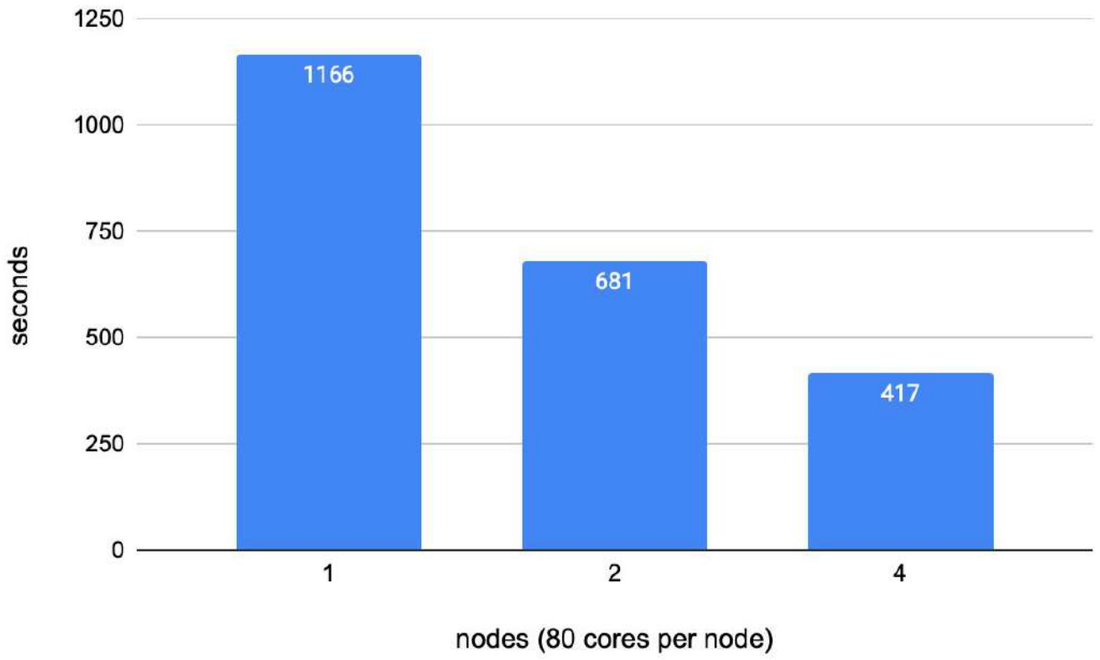


gcc	5.1.0
mpich	3.3.0
libxc	4.3.4
Blas	3.8.0
python	3.8.3

Total cores	Nodes	cores Per node	Time (second)
24	1	24	41130
48	2	24	7141
96	4	24	3085

Niagara using pure MPI parallelization

- **Multi-nodes has a good scalability on Niagara cluster.**
- **Using Intelmpi to parallelize, the performance is good.**

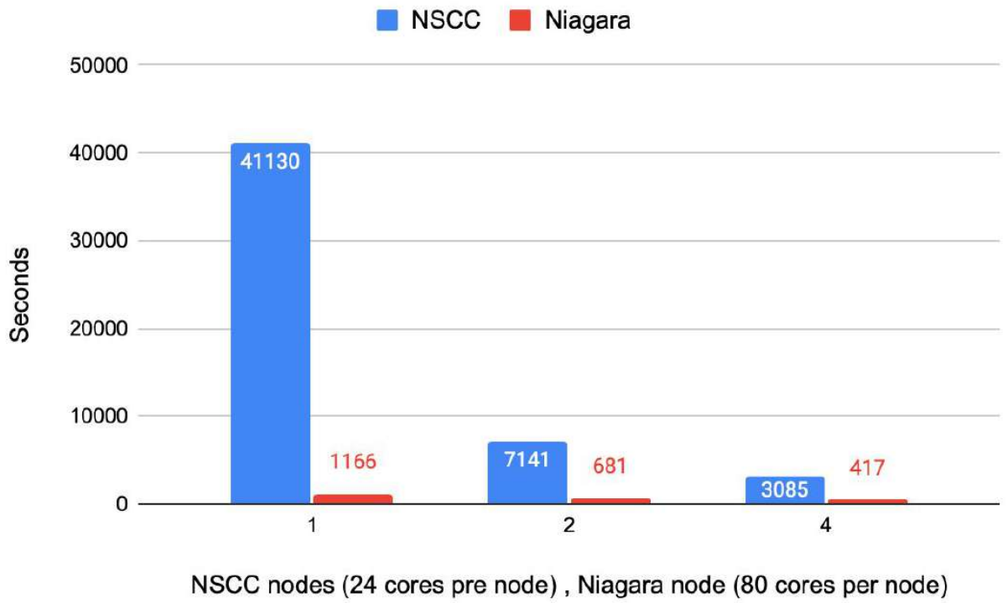


icc	2019u4
Intelmpi	2019u4
libxc	4.3.4
openblas	0.3.7
python	3.8.5

Total cores	Nodes	cores Per node	Time (second)
80	1	80	1166
160	2	80	681
320	4	80	417

NSCC vs. Niagara using pure MPI parallelization

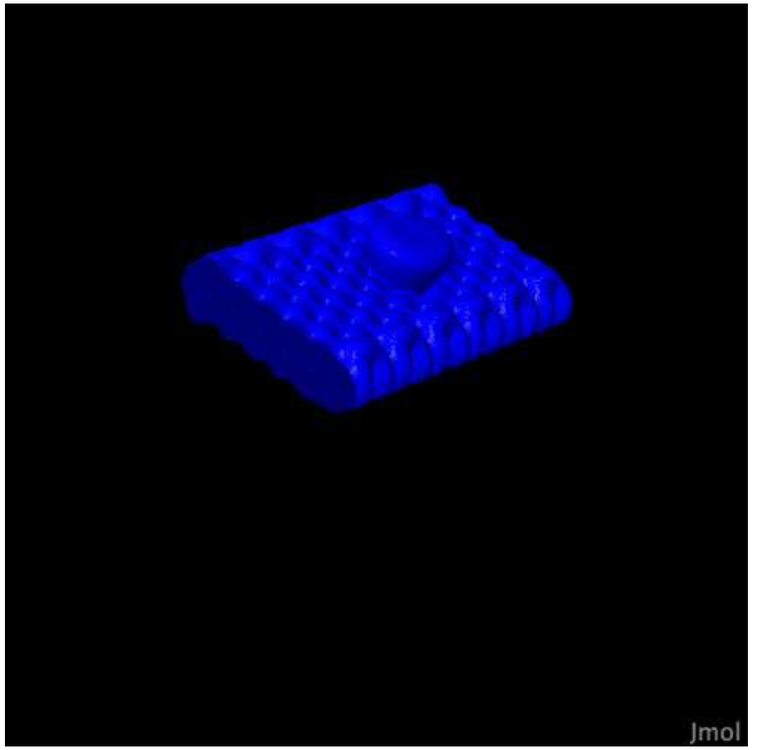
- Multi-nodes has a good scalability on both of Niagara cluster and NSCC cluster.
- Niagara cluster has more cores than NSCC cluster, but we can discover that performance of Intelmpi is better than mpich.



	NSCC	Niagara
C compiler	5.1.0	Intel 2019u4
mpi	mpich -3.3	Intelmpi 2019u4
libxc	4.3.4	4.3.4
Blas	3.8.0	No use
Openblas	No use	0.3.7
python	3.8.3	3.8.5

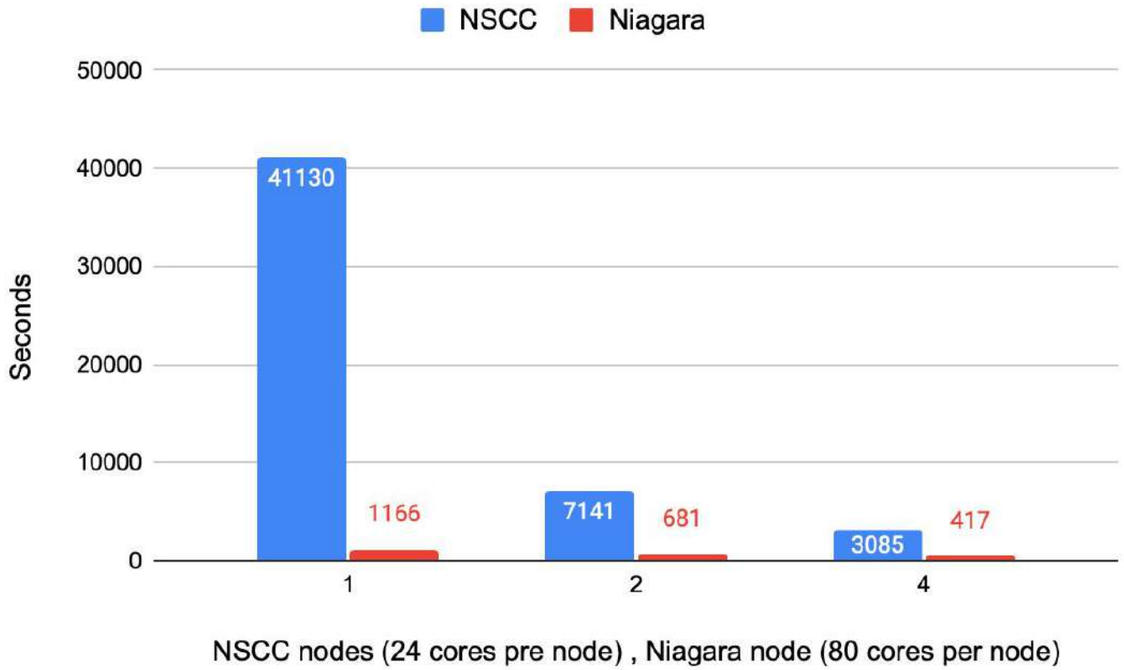
GPAW visualization

- Using jmol simulator to simulate the file elf_ribbon.cube file.
- One is image and the other is animation.



Choose which cluster to optimize.

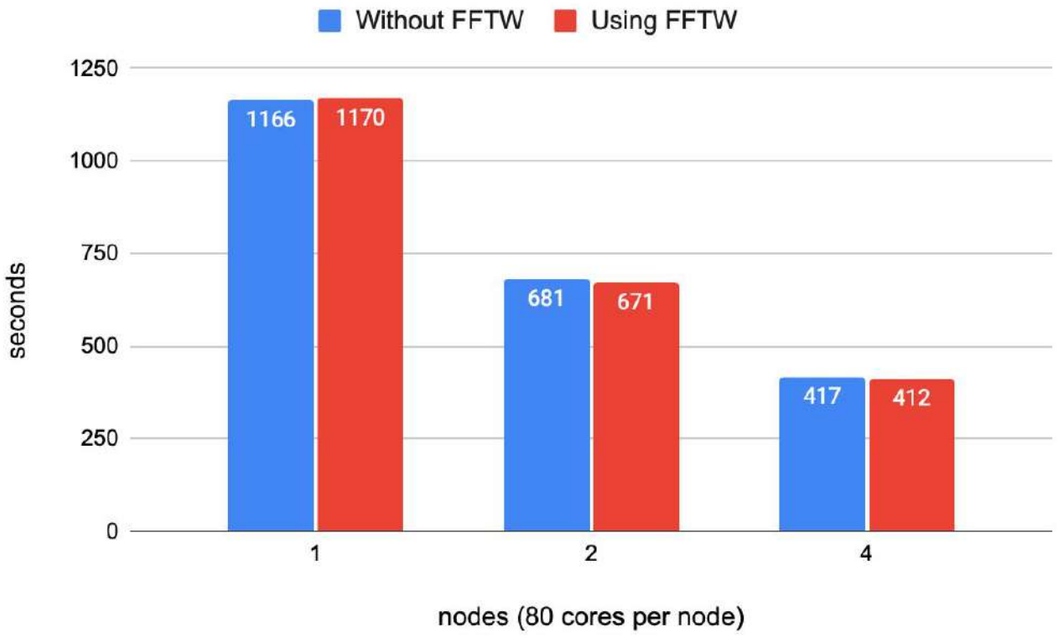
- When using four nodes and each node using all of cores per node to compare.
- The performance of Niagara cluster is much better than NSCC cluster.
- We choose Niagara cluster to optimize.



	NSCC	Niagara
Nodes	4	4
cores per node	24	80
Total cores	96	320
Time (seconds)	3085	417

FFTW

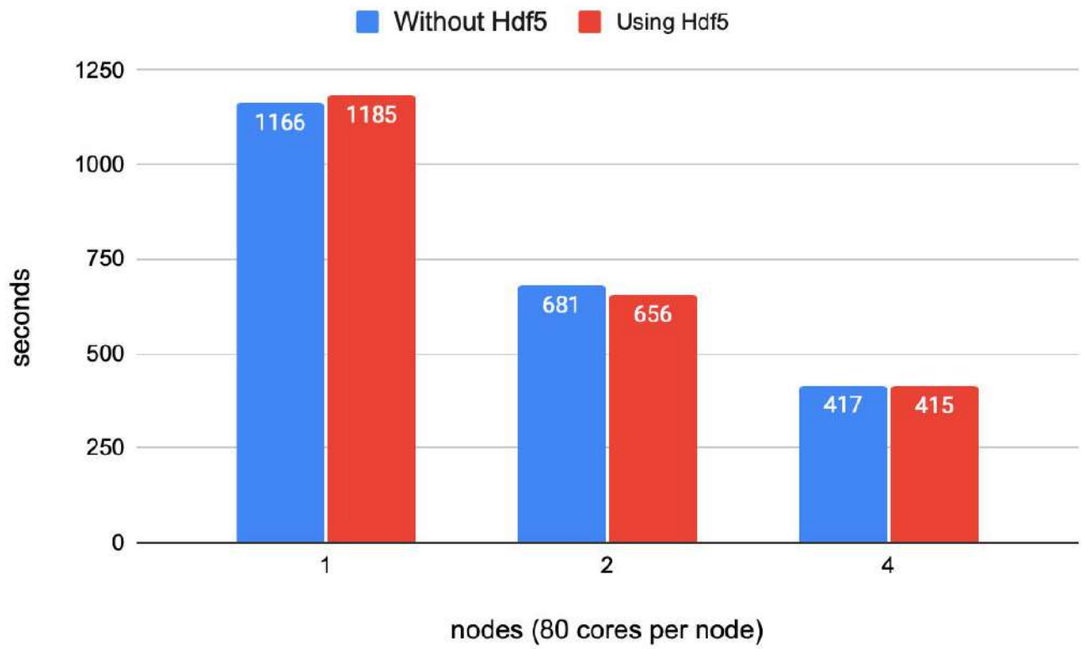
- **There is only a little speedup when using FFTW to optimize.**



cNode	Without FFTW (seconds)	Using FFTW (seconds)
1 (80 cores per node)	1166	1170
2 (80 cores per node)	681	671
4 (80 cores per node)	417	412

Hdf5

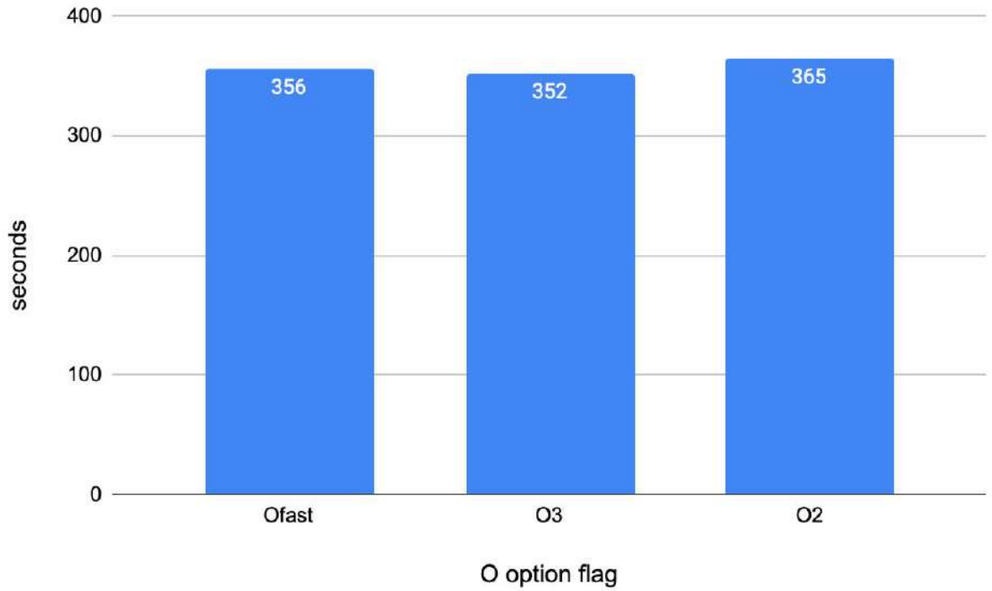
- **There is only a little speedup when using Hdf5 to optimize.**



Node	Without Hdf5	Using Hdf5
1 (80 cores per node)	1166	1185
2 (80 cores per node)	681	656
4 (80 cores per node)	417	415

- O option flag

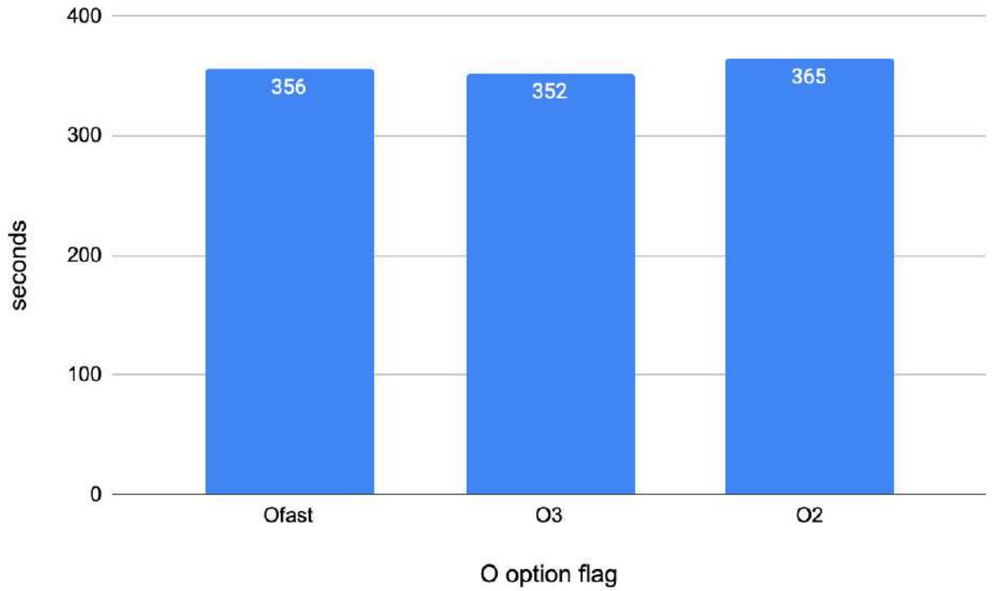
- Run on Niagara four nodes (80 cores per node).
- Speed : O3 > Ofast > O2
- The result of Ofast is correct, but the speed of O3 is fastest.



O option flag	Time (seconds)
Ofast	356
O3	352
O2	365

- O option flag

- Run on Niagara four nodes (80 cores per node).
- Speed : O3 > Ofast > O2
- The result of Ofast is correct, but the speed of O3 is fastest.



O option flag	Time (seconds)
Ofast	356
O3	352
O2	365

Vtune to find out the hotpot

- Using Intel vtune profiler to find out the hotspot.

Function Stack	CPU Time: Total ▼	CPU Time: Self	Module	Function (Full)	Source File	Start Address
Total	100.0%	0ms				
_start	60.0%	0ms	mpiex...	_start		0x4047c0
__libc_start_main	60.0%	0ms	libc.so.6	__libc_start_...		0x22460
main	60.0%	0ms	mpiex...	main	mpiexec.c	0x4048b0
mpiexec_get_parameters	50.0%	0ms	mpiex...	mpiexec_get...	mpiexec_...	0x41f270
i_set_default_ppn	30.0%	0ms	mpiex...	i_set_default...	i_mpiexec...	0x4201a6
i_read_default_env	20.0%	0ms	mpiex...	i_read_defau...	i_mpiexec...	0x422620
push_env_downstream	10.0%	0ms	mpiex...	push_env_d...	mpiexec.c	0x40ac40
_start	40.0%	0ms	srun	_start		0x406d70

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
openat	libc.so.6	0.040s
OS_BARESYSCALL_DoCallAsmIntel64Linux	libc-dynamic.so	0.020s
fscanf	libc.so.6	0.010s
pthread_create	libpthread.so.0	0.010s
__snprintf_chk	libc.so.6	0.010s
[Others]	libpin3dwarf.so	0.010s

*N/A is applied to non-summable metrics.

Hotspots Hotspots by CPU Utilization

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform | topology-linux.c

Function	CPU Time: Total — Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Overhead Time	CPU Time: Self	Module	
__libc_start_main	100.0%	0.0%	0.0%	0ms	libc.so.6	__libc_start_main
_start	60.0%	0.0%	0.0%	0ms	mpiexec.hydra	_start
main	60.0%	0.0%	0.0%	0ms	mpiexec.hydra	main
hwloc_look_linuxfs	50.0%	0.0%	0.0%	0ms	mpiexec.hydra	hwloc_look_linuxf
mpiexec_get_parameters	50.0%	0.0%	0.0%	0ms	mpiexec.hydra	mpiexec_get_para
ipl_entrance	50.0%	0.0%	0.0%	0ms	mpiexec.hydra	ipl_entrance
ipl_processor_info	50.0%	0.0%	0.0%	0ms	mpiexec.hydra	ipl_processor_info
ipl_detect_machine_topology	50.0%	0.0%	0.0%	0ms	mpiexec.hydra	ipl_detect_machin
hwloc_discover	50.0%	0.0%	0.0%	0ms	mpiexec.hydra	hwloc_discover
hwloc_topology_load	50.0%	0.0%	0.0%	0ms	mpiexec.hydra	hwloc_topology_l
main	40.0%	0.0%	0.0%	0ms	srun	main
_start	40.0%	0.0%	0.0%	0ms	srun	_start
look_sysfscpu	40.0%	0.0%	0.0%	0ms	mpiexec.hydra	look_sysfscpu
openat	40.0%	0.0%	0.0%	39.988ms	libc.so.6	openat
srun	30.0%	0.0%	0.0%	0ms	srun	srun
i_set_default_ppn	30.0%	0.0%	0.0%	0ms	mpiexec.hydra	i_set_default_ppn
hwloc__alloc_read_path_as_cpumask	30.0%	0.0%	0.0%	0ms	mpiexec.hydra	hwloc__alloc_rea
hwloc_open	30.0%	0.0%	0.0%	0ms	mpiexec.hydra	hwloc_open
hwloc__read_path_as_cpumask	30.0%	0.0%	0.0%	0ms	mpiexec.hydra	hwloc__read_path
OS_BARESYSCALL_DoCallAsmIntel64Lin	20.0%	0.0%	0.0%	20.014ms	libc-dynamic.so	OS_BARESYSCA
OS_SyscallDo	20.0%	0.0%	0.0%	0ms	libc-dynamic.so	OS_SyscallDo
i_read_default_env	20.0%	0.0%	0.0%	0ms	mpiexec.hydra	i_read_default_en
plugin_load_from_file	20.0%	0.0%	0.0%	0ms	libslurmfull.so	plugin_load_from_
plugin_context_create	20.0%	0.0%	0.0%	0ms	libslurmfull.so	plugin_context_cr
plugin_load_and_link	20.0%	0.0%	0.0%	0ms	libslurmfull.so	plugin_load_and_
dlopen	20.0%	0.0%	0.0%	0ms	libdl.so.2	dlopen
LoadDwarfForFile	20.0%	0.0%	0.0%	0ms	libpin3dwarf.so	LoadDwarfForFile
GetSubprogramsListInImage	20.0%	0.0%	0.0%	0ms	libpin3dwarf.so	GetSubprogramsL
list_for_each_max	20.0%	0.0%	0.0%	0ms	libslurmfull.so	list_for_each_ma

Scalable Python

- We reference the paper “Optimizing GPAW” try to use scalable python to optimize.
- We install a scalable version of python, but the version is based on python2 which couldn’t support GPAW 20.10.0.

Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

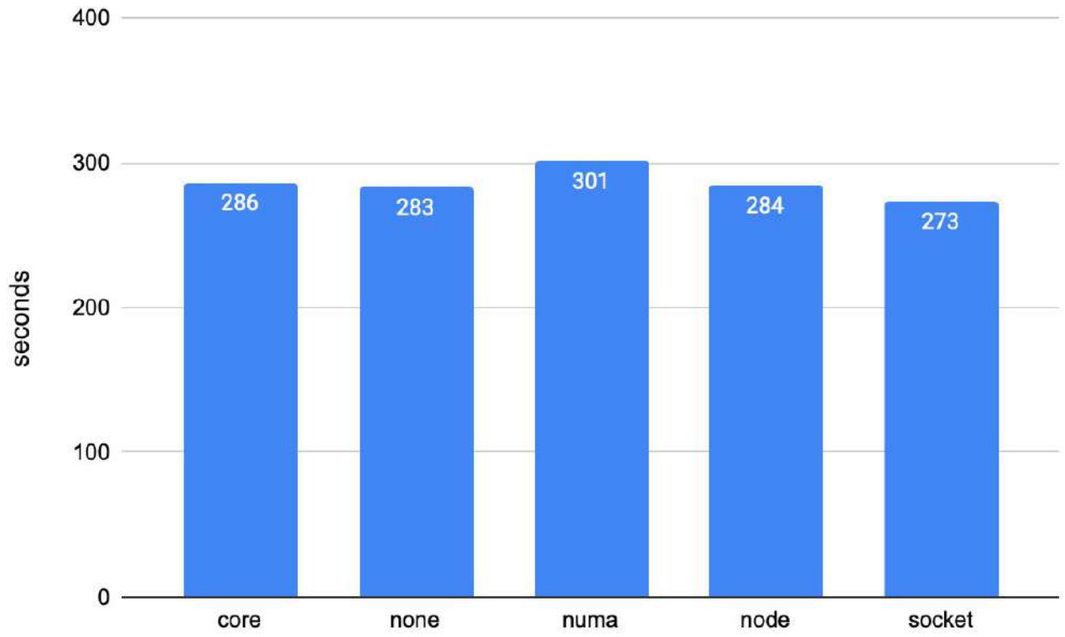
Optimizing GPAW

Jussi Enkovaara[a,*], Martti Louhivuori[a], Petar Jovanovic[b], Vladimir Slavnic[b], Mikael Rännar[c]

[a]CSC - IT Center for Science, P.O. Box 405 FI-02101 Espoo Finland
[b]Scientific Computing Laboratory, Institute of Physics Belgrade, Pregrevica 118, 11080 Belgrade, Serbia
[c]Department of Computing Science, Umea University, SE-901 87 Umea, Sweden

-bind-to flag

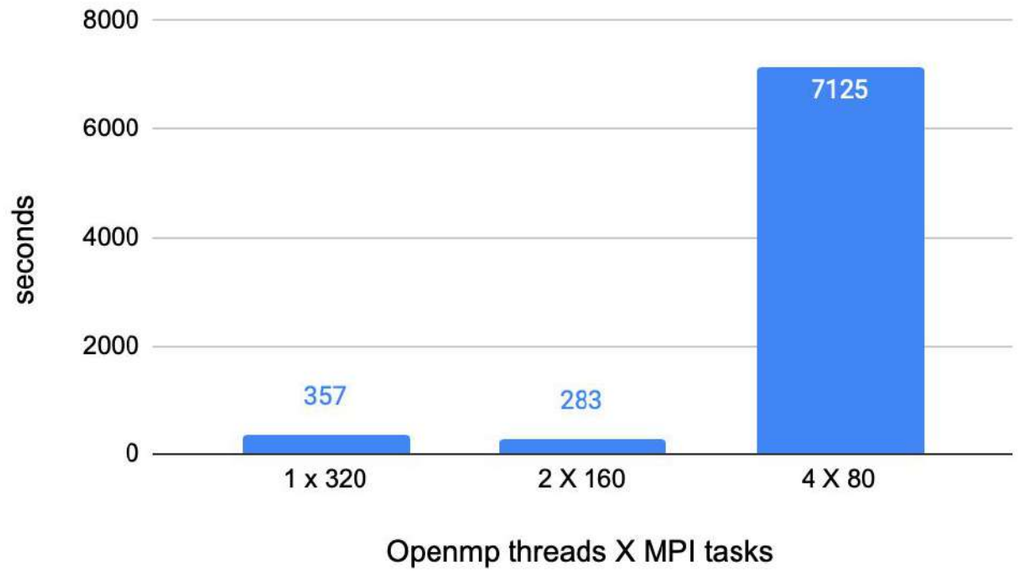
- Running GPAW on four node using different -bind-to flag.



-bind-to flag	Time (seconds)
core	286
none	283
node	284
numa	301
socket	273

Openmp threads

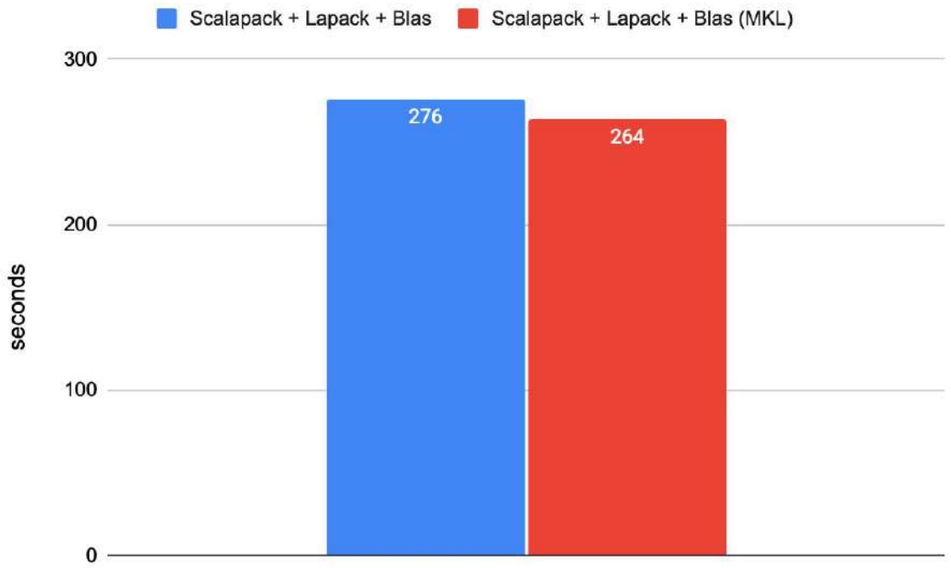
- **Using 320 cores to compare performance of different openmp threads number.**



	Threads	MPI tasks	Time (seconds)
1 X 320	1	320	357
2 X 160	2	160	283
4 X 80	4	80	7125

Scalapack, Lapack, and Blas with MKL

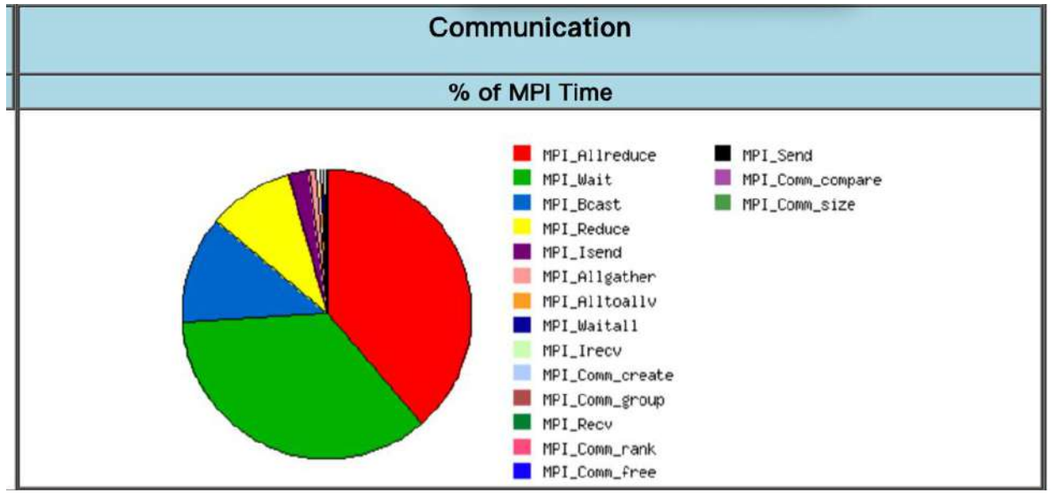
- Using MKL to optimize the math libraries.
- MKL make GPAW



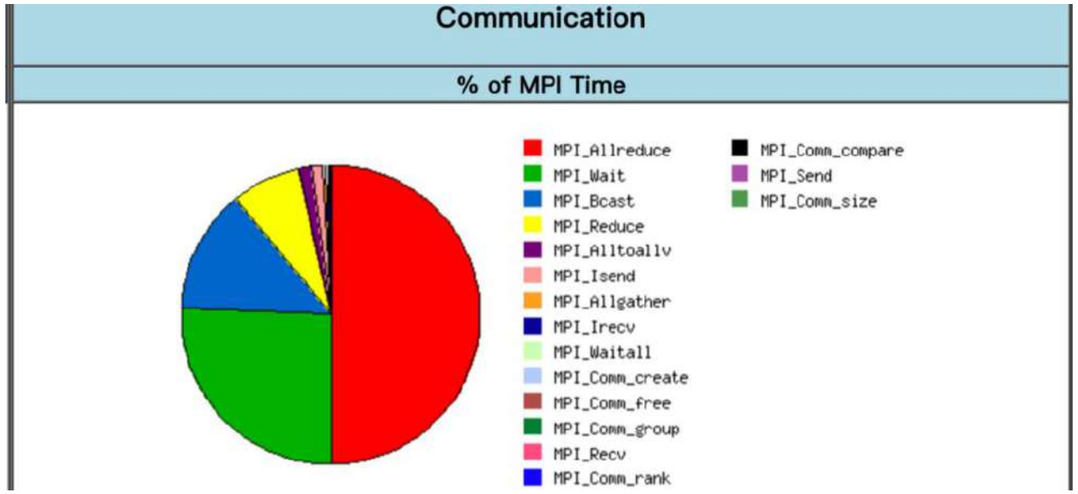
	Without MKL	Using MKL
Time (seconds)	276	264

Using IPM profiler to profile the

- When we run the GPAW on optimizing stage, we compare the both of profiling results to analysis.



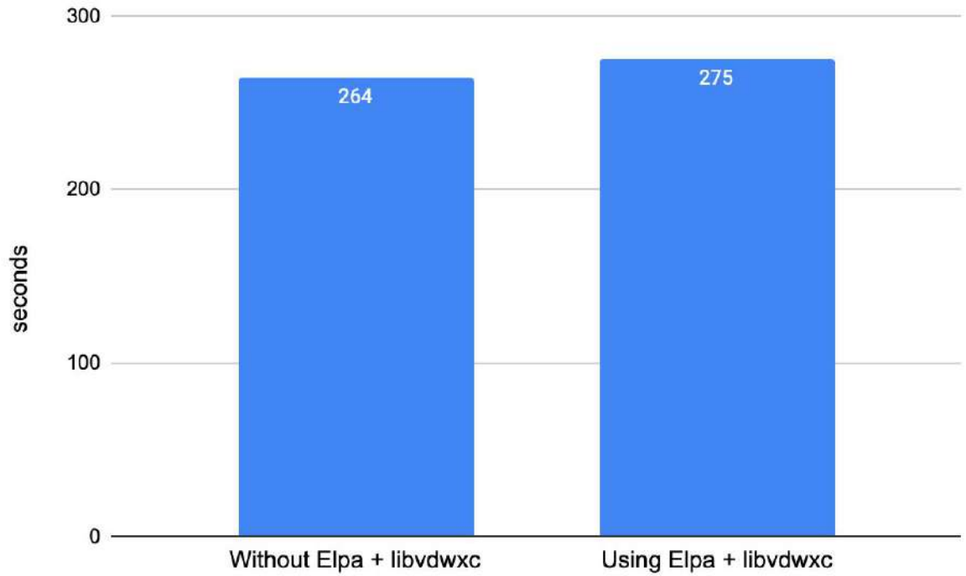
Before



After

Elpa & libvdwxc

- **Using Elap and libvdwxc to speedup the performance of GPAW.**



	Without Elpa + libvdwxc	Using Elpa + libvdwxc
Time (seconds)	264	275

Eigensolver and parallel runs option

- We try to use different eigensolver to the copper.py.
- Fix some parallel runs options.

Eigensolver	Time (seconds)
rmm-diis (default)	261
cg	upto 3600
dav	441

```
40
41 # setup parameters
42 args = {'h': h,
43         'nbands': -20,
44         'occupations': FermiDirac(0.2),
45         'kpts': kpts,
46         'xc': 'PBE',
47         'mixer': Mixer(0.1, 5, 100),
48         'eigensolver': RMMDIIS(),
49         'maxiter': maxiter,
50         'txt': txt,
51         'parallel' : {'band' : 2 ,'use_elpa': 'True', 'elpasolver' : '2stage'}
52         }
53
54 calc = GPAW(**args)
55 atoms.set_calculator(calc)
56
```

Final Result - Niagara Cluster

- **Software compilation version**
- **Hardware usage**

intel	2019u4
intelmpi	2019u4
openblas	0.3.7
libxc	4.3.4
python	3.8.5
Openmp	2019u4
fftw	mkl (intel 2019u4)
Scalapack	mkl (intel 2019u4)
Lapack	mkl (intel 2019u4)
Blas	mkl (intel 2019u4)

Blacs	mkl (intel 2019u4)
libvdwxc	0.4.0
Elpa	2021.05.001
Hdf5	1.8.21
O option flag	O3
-bind-to	socket
Openmp threads	2

Nodes	4
Total cores	320
Cores per node	80

Time: 00:04:24
= 264s

Speed up: 58%