TSVV - ACH meeting

The big picture

- Goal: To produce scientific codes that can use Tier-0 systems efficiently.
- GPUs are dominant at the top.
- What do we need to get there?

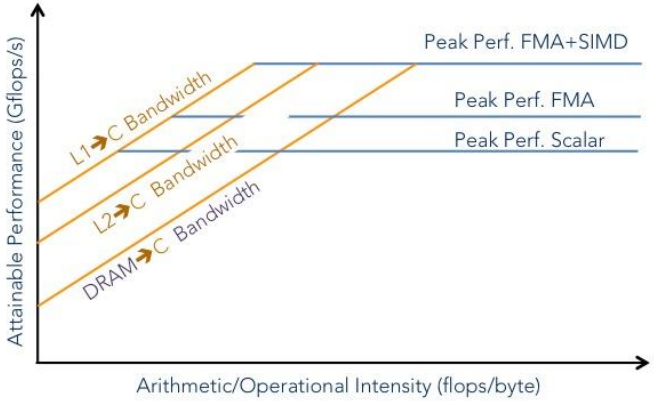
	Accelerator/Co-Processor	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
1	NVIDIA Tesla V100	80	16	243,448,930	475,572,809	5,059,976
2	NVIDIA A100	15	3	226,001,000	324,135,290	2,125,952
3	NVIDIA Tesla V100 SXM2	12	2.4	91,975,490	182,486,069	2,059,208
4	NVIDIA Tesla P100	8	1.6	49,751,640	73,680,456	1,005,472
5	NVIDIA A100 SXM4 40 GB	5	1	81,312,000	115,202,938	869,192
6	NVIDIA A100 40GB	4	0.8	30,133,600	47,814,630	315,812
7	NVIDIA Volta GV100	4	0.8	269,439,000	362,564,722	4,408,096
8	NVIDIA Tesla K40	3	0.6	8,824,090	14,612,320	201,328
9	NVIDIA A100 80GB	2	0.4	13,806,000	18,688,410	124,160
10	Matrix-2000	1	0.2	61,444,500	100,678,664	4,981,760
11	NVIDIA 2050	1	0.2	2,566,000	4,701,000	186,368
12	NVIDIA Tesla K40m	1	0.2	2,478,000	4,946,790	64,384
13	NVIDIA Tesla K40/Intel Xeon Phi 7120P	1	0.2	3,126,240	5,610,481	152,692
14	NVIDIA Tesla P100 NVLink	1	0.2	8,125,000	12,127,069	135,828
15	Preferred Networks MN-Core	1	0.2	1,822,400	3,137,870	1,664
16	Nvidia Volta V100	1	0.2	21,640,000	29,354,000	347,776
17	NVIDIA Tesla K80	1	0.2	2,592,000	3,798,600	66,000
18	Intel Xeon Phi 31S1P	1	0.2	2,071,390	3,074,534	174,720
19	Deep Computing Processor	1	0.2	4,325,000	6,134,170	163,840
20	AMD Vega 20	1	0.2	1,661,000	2,472,960	100,800
21	Intel Xeon Phi 5110P	1	0.2	2,539,130	3,388,032	194,616
22	NVIDIA Tesla K20x	1	0.2	3,188,000	4,605,000	72,000

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, **Fujitsu** RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, **IBM** DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, **IBM / NVIDIA / Mellanox** DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, **NRCPC** National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Perlmutter** - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, **HPE** DOE/SC/LBNL/NERSC United States	706,304	64,590.0	89,794.5	2,528
6	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, **Nvidia** NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
7	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, **NUDT** National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
8	**JUWELS Booster Module** - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, **Atos** Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
9	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, **Dell EMC** Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
10	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, **Dell EMC** Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	

HPC first principles - Roofline model and memory wall

HPC means reducing Time To Solution (more science, increased accuracy, less energy used…)

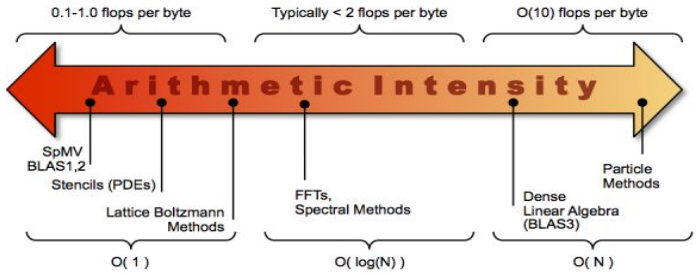
Roofline model: (if latencies are covered) **TTS is proportional to the inverse of memory** or **arithmetic throughput (GB/s or Gflops/s)**, which is determined by the arithmetic intensity of your algorithm (and the memory wall).



•Total volume communication across all memory hierarchies relatively to the core:

Attainable perf Gflops/s = min { Peak performace Gflops/s ; Bandwidths to Core x Arithmetic Intensity }

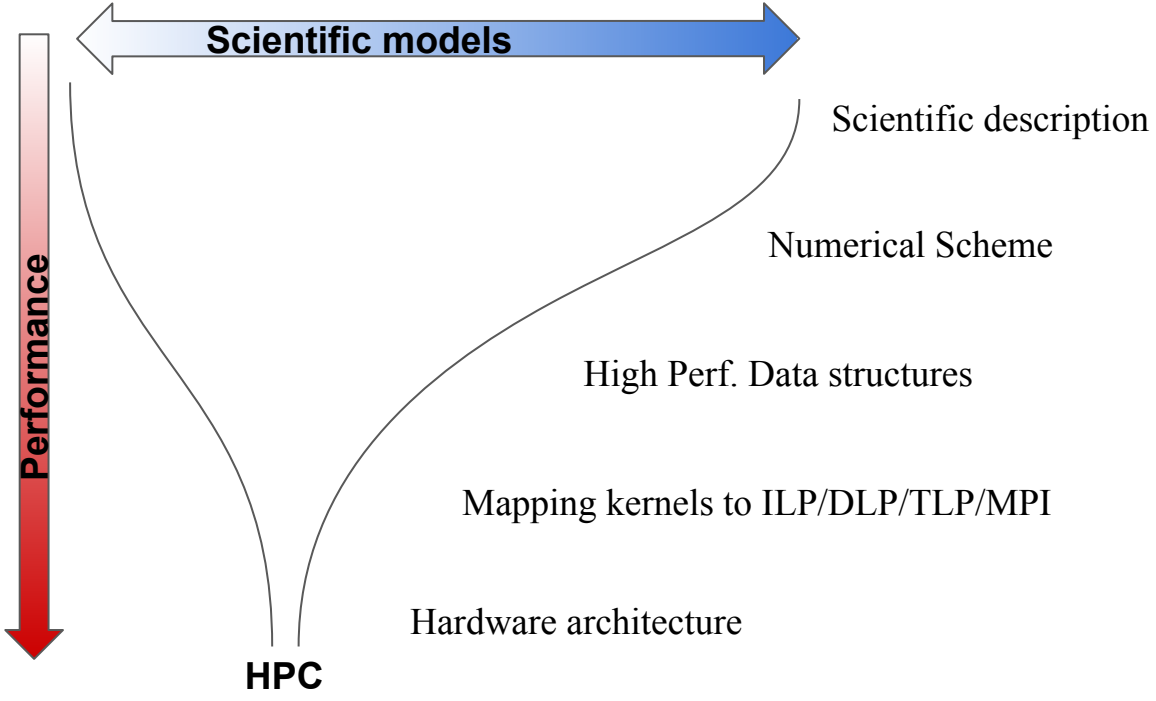
Bandwidths to Core { Bandwidth from L1 to Core ; Bandwidth from L2 to Core ; Bandwidth from L3 to Core ; Bandwidth from DRAM to Core }



Little's law: throughput = **parallelism**/latency or 1/throughput = latency/**parallelism** ~ TTS

Reducing TTS means exposing parallelism in scientific software

Research Codes vs. HPC codes: Scope



Two fluid drift-reduced Braginskii equations, $k_\perp^2 \gg k_\parallel^2$, $d/dt \ll \omega_{ci}$

$$\frac{\partial n}{\partial t} = -\frac{1}{B}[\phi, n] + \frac{2}{eB}[C(p_\mathrm{e}) - enC(\phi)] - \nabla_\parallel(nv_{\parallel\mathrm{e}}) + \mathcal{D}_\mathrm{n}(n) + S_\mathrm{n} + n_\mathrm{n}\nu_\mathrm{iz} - n\nu_\mathrm{rec} \quad (1)$$

$$\frac{\partial \tilde{\omega}}{\partial t} = -\frac{1}{B}[\phi, \tilde{\omega}] - v_{\parallel\mathrm{i}}\nabla_\parallel\tilde{\omega} + \frac{B^2}{mn}\nabla_\parallel j_\parallel + \frac{2B}{mn}C(p) + \mathcal{D}_{\tilde{\omega}}(\tilde{\omega}) - \frac{n_\mathrm{n}}{n}\nu_\mathrm{cx}\tilde{\omega} \quad (2)$$

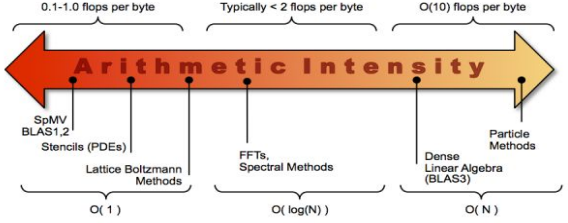
$$\frac{\partial v_{\parallel\mathrm{e}}}{\partial t} + \frac{e}{m_\mathrm{e}}\frac{\partial \Psi}{\partial t} = -\frac{1}{B}[\phi, v_{\parallel\mathrm{e}}] - v_{\parallel\mathrm{e}}\nabla_\parallel v_{\parallel\mathrm{e}} + \frac{e}{\sigma_\parallel m_\mathrm{e}}j_\parallel + \frac{e}{m_\mathrm{e}}\nabla_\parallel\phi - \frac{T_\mathrm{e}}{m_\mathrm{e}n}\nabla_\parallel n - \frac{1.71}{m_\mathrm{e}n}\nabla_\parallel T_\mathrm{e} + \mathcal{D}_{v_{\parallel\mathrm{e}}}(v_{\parallel\mathrm{e}}) + \frac{n_\mathrm{n}}{n}(\nu_\mathrm{en} + 2\nu_\mathrm{iz})(v_{\parallel\mathrm{n}} - v_{\parallel\mathrm{e}}) \quad (3)$$

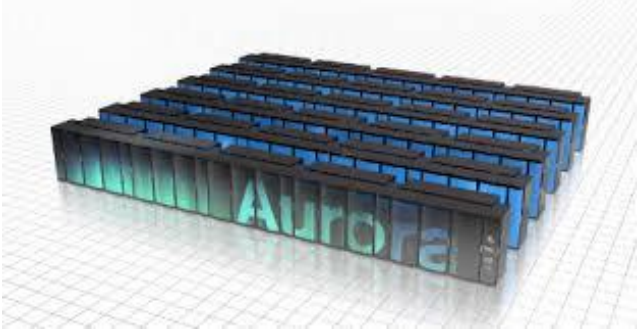
$$\frac{\partial v_{\parallel\mathrm{i}}}{\partial t} = -\frac{1}{B}[\phi, v_{\parallel\mathrm{i}}] - v_{\parallel\mathrm{i}}\nabla_\parallel v_{\parallel\mathrm{i}} - \frac{1}{m_\mathrm{i}n}\nabla_\parallel p + \mathcal{D}_{v_{\parallel\mathrm{i}}}(v_{\parallel\mathrm{i}}) + \frac{n_\mathrm{n}}{n}(\nu_\mathrm{iz} + \nu_\mathrm{cx})(v_{\parallel\mathrm{n}} - v_{\parallel\mathrm{i}}) \quad (4)$$

$$\frac{\partial T_\mathrm{e}}{\partial t} = -\frac{1}{B}[\phi, T_\mathrm{e}] - v_{\parallel\mathrm{e}}\nabla_\parallel T_\mathrm{e} + \frac{4T_\mathrm{e}}{3eB}\left[\frac{T_\mathrm{e}}{n}C(n) + \frac{7}{2}C(T_\mathrm{e}) - eC(\phi)\right] + \frac{2T_\mathrm{e}}{3n}\left[\frac{0.71}{e}\nabla_\parallel j_\parallel - n\nabla_\parallel v_{\parallel\mathrm{e}}\right] + \mathcal{D}_{T_\mathrm{e}}(T_\mathrm{e}) + \mathcal{D}_{T_\mathrm{e}}^{\parallel}(T_\mathrm{e}) + S_{T_\mathrm{e}} + \frac{n_\mathrm{n}}{n}\nu_\mathrm{iz}\left[-\frac{2}{3}E_\mathrm{iz} - T_\mathrm{e} + m_\mathrm{e}v_{\parallel\mathrm{e}}\left(v_{\parallel\mathrm{e}} - \frac{4}{3}v_{\parallel\mathrm{n}}\right)\right] - \frac{n_\mathrm{n}}{n}\nu_\mathrm{en}m_\mathrm{e}\frac{2}{3}v_{\parallel\mathrm{e}}(v_{\parallel\mathrm{n}} - v_{\parallel\mathrm{e}}) \quad (5)$$

$$\frac{\partial T_\mathrm{i}}{\partial t} = -\frac{1}{B}[\phi, T_\mathrm{i}] - v_{\parallel\mathrm{i}}\nabla_\parallel T_\mathrm{i} + \frac{4T_\mathrm{i}}{3eB}\left[C(T_\mathrm{e}) + \frac{T_\mathrm{e}}{n}C(n) - \frac{5}{3}C(T_\mathrm{i}) - eC(\phi)\right] + \frac{2T_\mathrm{i}}{3n}\left[\frac{1}{e}\nabla_\parallel j_\parallel - n\nabla_\parallel v_{\parallel\mathrm{i}}\right] + \mathcal{D}_{T_\mathrm{i}}(T_\mathrm{i}) + \mathcal{D}_{T_\mathrm{i}}^{\parallel}(T_\mathrm{i}) + S_{T_\mathrm{i}} + \frac{n_\mathrm{n}}{n}(\nu_\mathrm{iz} + \nu_\mathrm{cx})\left[T_\mathrm{n} - T_\mathrm{i} + \frac{1}{3}(v_{\parallel\mathrm{n}} - v_{\parallel\mathrm{i}})^2\right] \quad (6)$$

$$\rho_* = \rho_\mathrm{s}/R,\ \nabla_\parallel f = \mathbf{b}_0 \cdot \nabla f,\ \tilde{\omega} = \omega + \tau\nabla_\perp^2 T_\mathrm{i},\ p = n(T_\mathrm{e} + \tau T_\mathrm{i})$$

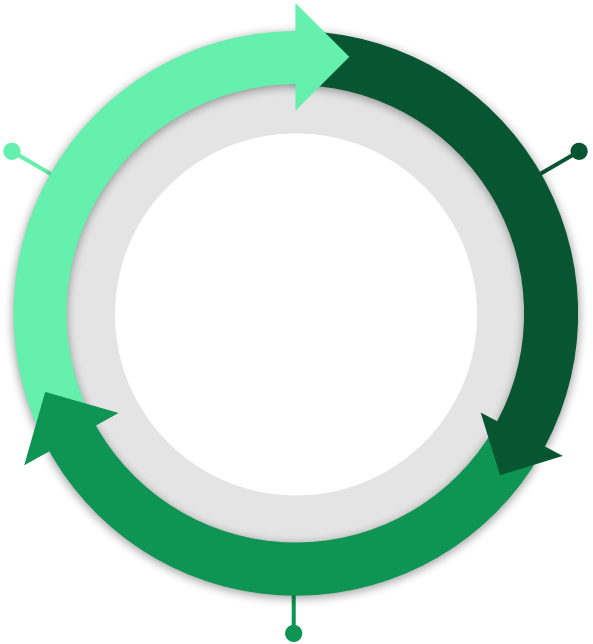




How do we make sure to reach maximum performance?

How to transition towards HPC codes

Miniapp injection back into the main codes



- Benchmarks definition and strategy.
- Profiling.
- Identification of bottlenecks.
- Translation into computational patterns.

MINIAPPS

- Improvements of specific patterns.
- Performance analysis in-depth.

Codes and bottlenecks - benchmarks strategy

- Goal: To understand the application’s behavior as a function of the resources.
- Many HPC call for resources requires representative benchmarks showing performance/scalability of the application.
- Usually this activity is performed “by hand”.
- From last meeting: “It would be interesting to introduce more automation in the benchmarking activity”.

Towards automatic benchmarks: Idea

Benchmark generation → Benchmark postprocessing → Analysis

- We must select representative test cases.
- A test case can be executed with different number of resources.
- Profilers can help to understand the behavior of the application.

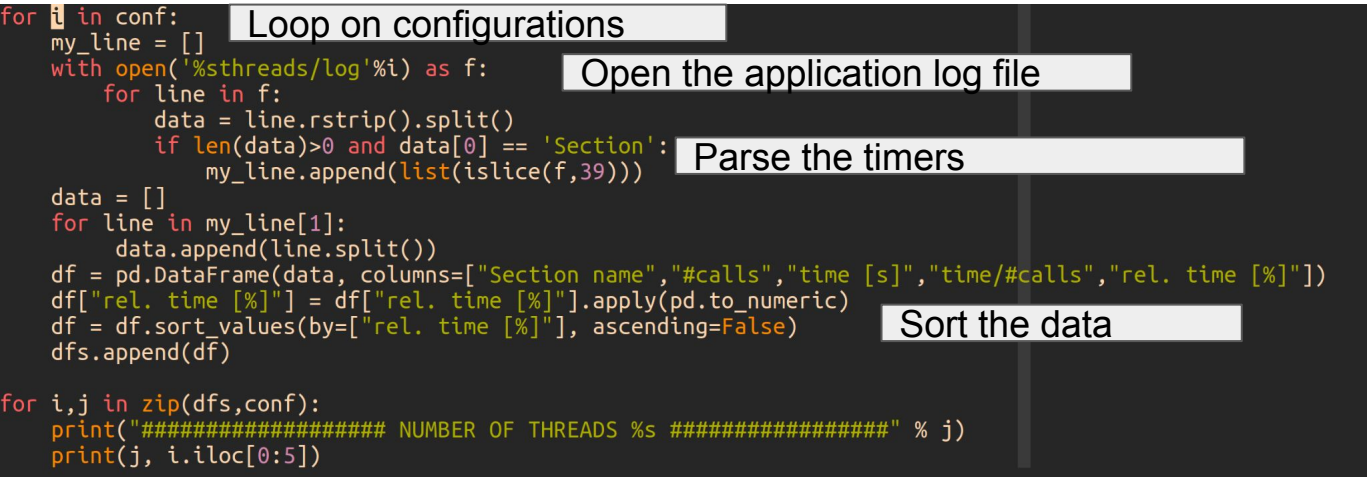
- Many profilers have a cli interface that allows to extract information.
- We can also parse timers from the applications.

- What can be optimized?

Towards automatic benchmarks - generation

- The configurations represent the resources (threads, MPI)
- The configuration are application and machine dependent.

Towards automatic benchmarks - analysis of the application log




```
for i in conf:
    my_line = []
    with open('%sthreads/log'%i) as f:
        for line in f:
            data = line.rstrip().split()
            if len(data)>0 and data[0] == 'Section':
                my_line.append(list(islice(f,39)))
    data = []
    for line in my_line[1]:
        data.append(line.split())
    df = pd.DataFrame(data, columns=["Section name","#calls","time [s]","time/#calls","rel. time [%]"])
    df["rel. time [%]"] = df["rel. time [%]"].apply(pd.to_numeric)
    df = df.sort_values(by=["rel. time [%]"], ascending=False)
    dfs.append(df)

for i,j in zip(dfs,conf):
    print("################### NUMBER OF THREADS %s #################" % j)
    print(j, i.iloc[0:5])
```

- If the application has timers it is possible to extract them.
- Then, we can sort the timers in ascending order.
- And, by changing the resources.

Towards automatic benchmarks - analysis of the profiler log

```
b = []
for i in number_of_threads:
    string = str(i) + 'threads'
    os.system("amplxe-cl -report hotspots -r %s/profile -format=csv \
    -report-output=hotspots.out -csv-delimiter=',' " % string)
    a = pd.read_csv('hotspots.out')
    c = a[['Function','CPU Time']][0:10]
    #print(c)
    c.insert(0, 'Threads', np.array([i for j in range(0,10)],dtype=np.int))
    b.append(c)
a = pd.DataFrame(b)
a.to_excel('hotspots.xlsx')
```

Loop on configurations

Extract the profiler log

Save the data

- We can also use profilers to get more information on the application bottlenecks.

Knowing the tool: VTUNE

Elapsed Time: 97.672s

CPU Time:	1454.760s
Effective Time:	1398.894s
Spin Time:	54.314s
Imbalance or Serial Spinning:	0.168s
Lock Contention:	0s
MPI Busy Wait Time:	54.137s
Other:	0.010s
Overhead Time:	1.552s
Total Thread Count:	20
Paused Time:	0s

1 THREAD

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
bspline_sub_module_mp_dbvalu_	grillix	274.941s
csrmv_single	grillix	240.103s
polygon_m_mp_dist_to_polyedge_	grillix	96.724s
pmpi_allreduce_	libmpifort.so.12	78.927s
bspline_sub_module_mp_dintrv_	grillix	59.747s
[Others]		704.318s

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

- Tools like VTUNE provide a lot of information.

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

Elapsed Time: 184.023s

CPU Time:	5166.610s
Effective Time:	2136.268s
Spin Time:	2264.425s
Imbalance or Serial Spinning:	1997.476s
Lock Contention:	0.328s
MPI Busy Wait Time:	96.815s
Other:	169.806s
Overhead Time:	765.916s
Total Thread Count:	36
Paused Time:	0s

2 THREADS

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
__kmp_fork_barrier	libiomp5.so	2149.705s
pmpi_allreduce_	libmpifort.so.12	654.432s
__kmp_fork_call	libiomp5.so	496.909s
bspline_sub_module_mp_dbvalu_	grillix	268.085s
_INTERNAL_25_____src_kmp_runtime_cpp_337ef511::__kmp_itt_stack_callee_enter	libiomp5.so	264.690s
[Others]		1332.788s

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

OVERVIEW OF THE CODES

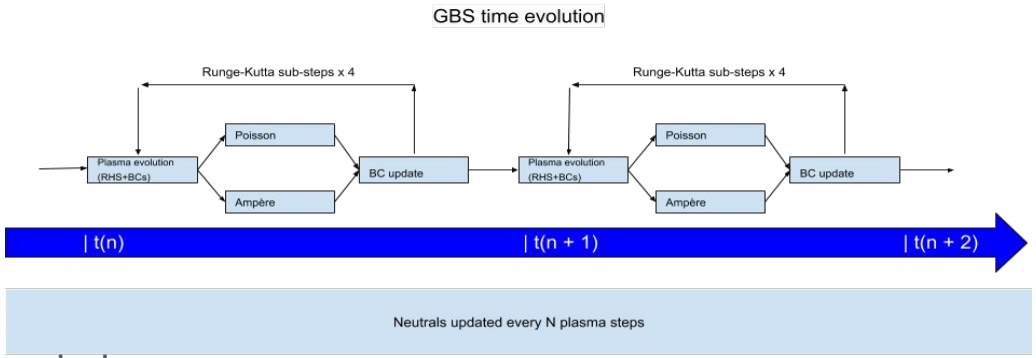
- GRILLIX
- GBS
- SOLEDGE3X

GRILLIX

- Current status:
  - Compilation:
    - Dependencies: cmake, mpi, openmp, hdf5, lapack, blas, netcdf
    - Tested on Izar@EPFL and Marconi: Worked out of the box.
  - Parallelization:
    - The poloidal planes are distributed with MPI
    - Each poloidal plane is parallelized with OpenMP
- Benchmark strategy:
  - The number of poloidal planes is fixed and distributed with MPI
  - For each poloidal plane is possible to increase the number of threads
  - Goal: To study the scalability of GRILLIX by increasing the number of threads, up to 1 MPI per NUMA socket.

GBS

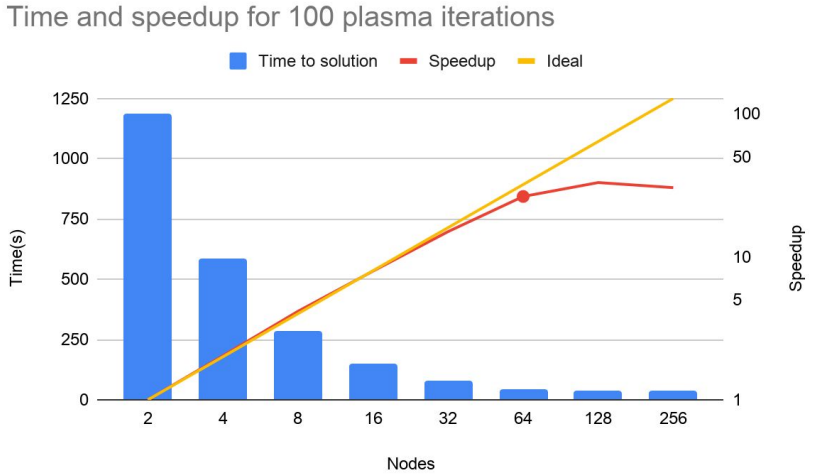
- Current status:
  - Compilation:
    - Dependencies: cmake, mpi, openmp, hdf5, lapack, blas, petsc, hypre, AMGX
    - Successfully tested on a variety of architectures, including GPUs.
  - Solver:
    - Direct(MUMPS)
    - Iterative(PETSc, AMGX)
    - Parallelization:
      - MPI in the poloidal plane
  - RHS computation:
    - Stencil operations - memory bounded
    - Parallelization: MPI across the domain



- Benchmark strategy:
  - We usually increase the number of MPI task on the toroidal direction and fix the number of MPI tasks in the poloidal plane

TCV on SuperMUC-NG

Nodes	Time to solution	Speedup	Grad comp	Ampere	Poisson
2	1187.72	1.00	361.39	562.66	262.3
4	584.54	2.03	163.46	286.49	133.53
8	285.58	4.16	72.79	144.02	67.32
16	149.45	7.95	36.64	75.67	35.05
32	79.17	15.00	21.88	37.98	17.74
64	44.96	26.42	14.9	19.42	9.29
128	35.95	33.04	8.19	19.05	7.78
256	39.01	30.45	5.37	25.98	6.84



SuperMUC-NG hardware

- 6480 Intel Skylake Xeon Platinum 8174
- 96GB DDR4 memory
- Network: Intel Omni-Path

Setup: TCV at 0.9T, 100 timestep

- Turbulent mode
- Nx = 300, ny = 600, nz = 128
- Solver: deflated GMRES
- Preconditioner: Hypre/BoomerAMG

x-y-z processor partition

2 node: 6x8x2 cores
4 nodes: 6x8x4 cores
8 nodes: 6x9x8 cores
16 nodes: 6x8x16 cores
32 nodes: 6x8x32 cores
64 nodes: 6x8x64 cores
128 nodes: 6x16x64 cores
256 nodes: 12x16x64 cores

AT **64 NODES** IT TAKES **0.45s/STEP TO SOLVE FULL TCV → ~6M CORE HOURS** FOR THE FULL SIMULATION

JT60-SA on SuperMUC-NG

Nodes	System	Time to solution	Plasma	Ampere	Poisson
64	TCV	44.96	14.9	19.42	9.29
75	JT60-SA	909.93	501.19	196.78	199.6

SuperMUC-NG hardware
- 6480 Intel Skylake Xeon Platinum 8174
- 96GB DDR4 memory
- Network: Intel Omni-Path

Setup: JT60-SA, 100 timestep
- Turbulent mode
- Nx = **1200**, ny = **2000**, nz = 300
- Solver: deflated GMRES
- Preconditioner: Hypre/BoomerAMG

9(s/step)*10M(steps)*75(nodes)*48(cores/node) ~ 90M core/hours

System	Architecture	Site (Country)	Core Hours (node hours)	Minimum request (core hours)
HAWK	HPE Apollo	GCS@HLRS (DE)	345,6 million (2.7 million)	100 million
Joliot-Curie KNL	BULL Sequana X1000	GENCI@CEA (FR)	37.5 million (0.6 million)	15 million
Joliot-Curie Rome	BULL Sequana XH2000	GENCI@CEA (FR)	195.3 million (1.5 million)	15 million
Joliot-Curie SKL	BULL Sequana X1000	GENCI@CEA (FR)	52.9 million (1.1 million)	15 million
JUWELS Booster	BULL Sequana XH2000	GCS@JSC (DE)	35.04 million (0.73 million)	5.76 million Use of GPUs
JUWELS Cluster	BULL Sequana X1000	GCS@JSC (DE)	70 million (1.46 million)	35 million
Marconi100	IBM Power 9 AC922 Whiterspoon	CINECA (IT)	660 million (1.875 million)	35 million Use of GPUs
MareNostrum 4	Lenovo System	BSC (ES)	240 million (5.0 million)	30 million
Piz Daint	Cray XC50 System	ETH Zurich/CSCS (CH)	510 million (7.5 million)	68 million Use of GPUs
SuperMUC-NG	Lenovo ThinkSystem	GCS@LRZ (DE)	123 million (2.56 million)	35 million

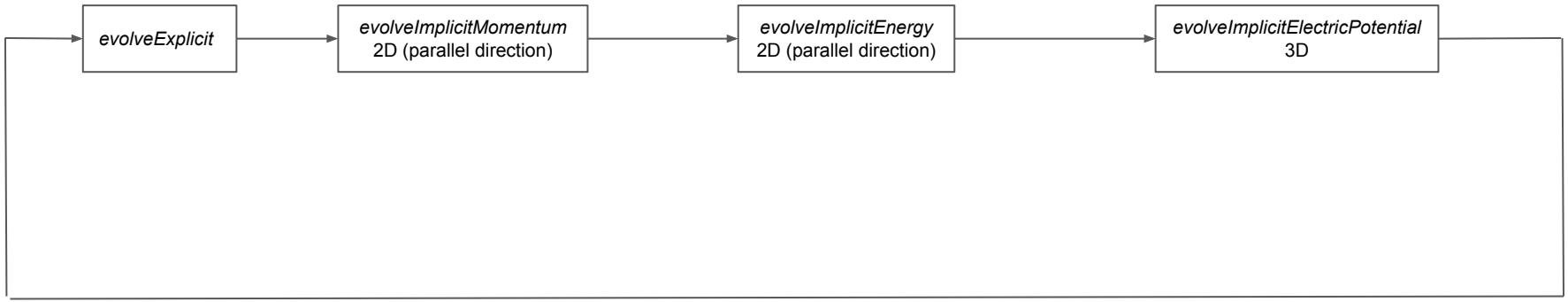
Soledge3X profiling

- **Current status:**
  - Soledge3X relies on a mix explicit-implicit scheme
  - It uses MPI+OpenMP
  - it uses Petsc, Pastix, Hypre for implicit solvers

- **Goals:**
  - profiling Soledge3X on SCITAS and Marconi clusters
  - implement performance metrics to understand main bottlenecks
  - use miniapp to investigate main bottlenecks and analyse performance in depth
  - optimize and porting to GPU some parts of the code

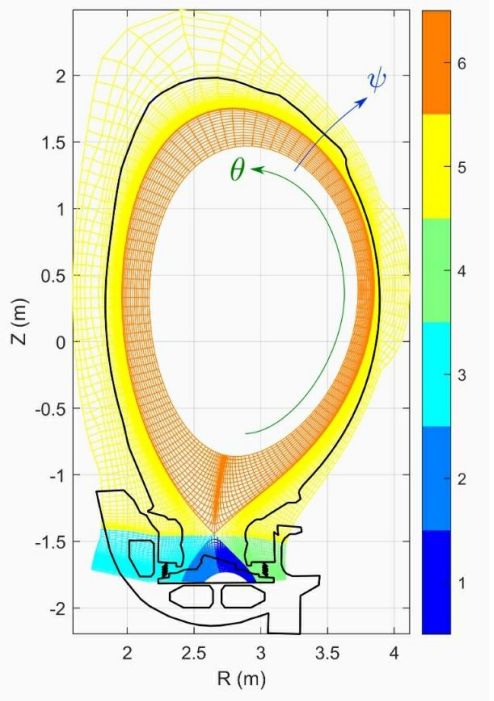
Time-stepping scheme in Soledge3X

- Main loop algorithm regarding main CPU time-consuming routines



Parallelization in Soledge3X

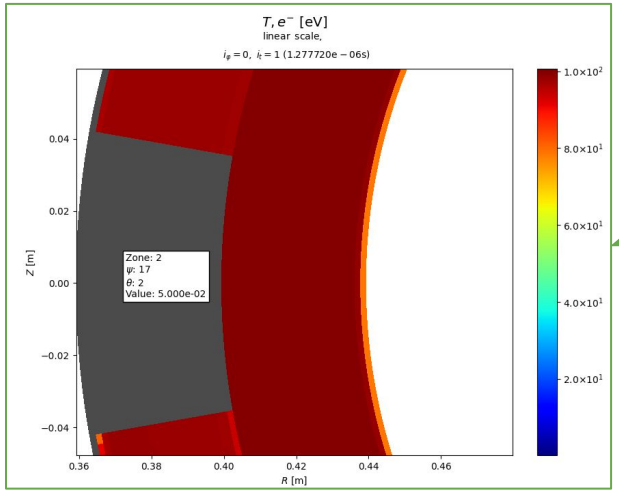
- Spatial discretization:
  - structured grid in the (ψ,θ,φ) coordinate system aligned with magnetic flux surfaces (ψ associated with the magnetic flux)
  - the solvers *evolveImplicitMomentum* and *evolveImplicitEnergy* are built using 2D stencils located in magnetic flux surface:
    → independent linear 2D mesh-based solvers are called for each value of ψ (magnetic flux surface)
  - however, the solver *evolveImplicitElectricPotential* is 3D mesh-based
- PETSC, PASTIX and HYPRE can be used for implicit solvers
- The domain is decomposed in zones for X-point geometries (see figure)
- MPI domain decomposition according to the (ψ,θ,φ) structured grid: the domain is in priority decomposed along the ψ direction (according to the magnetic flux surface workload), then along the θ direction
- MPI communicator for each magnetic flux surface (each value of ψ), useful for 2D mesh-based solvers
- OpenMP is used for each MPI process, except in PETSC and HYPRE solvers

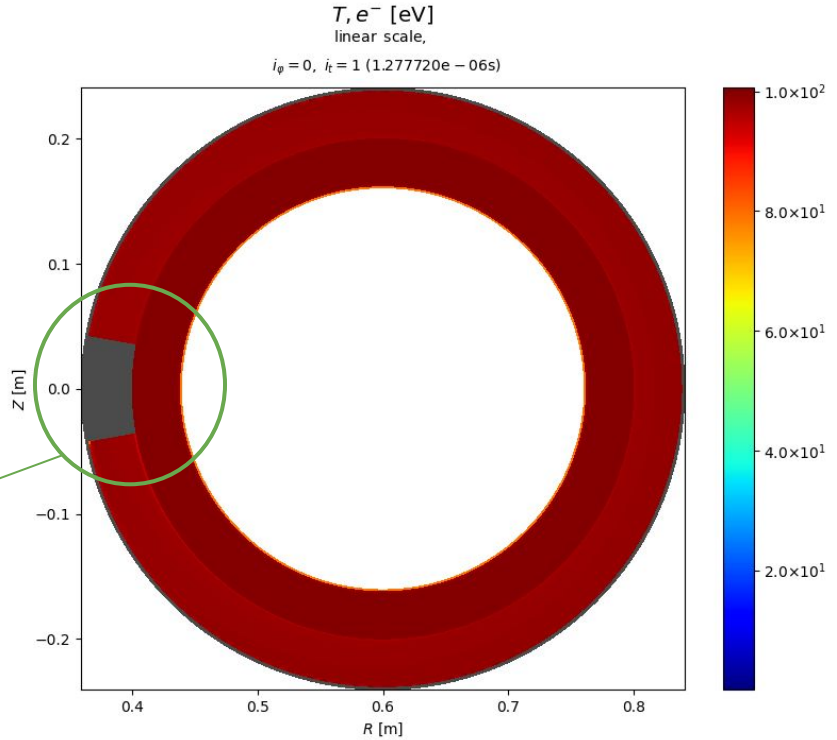


Profiling setup

- Setup: Helvetios@SCITAS cluster
  - 2 Skylake processors running at 2.3 GHz, with 18 cores each
  - 192 GB of DDR3 RAM
  - Intel compiler

- Test case : circle 3D
  - Npsi = 50, Ntheta = 500, Nphi = 50
  - Petsc for all implicit solvers
    - BiCGStab (Stabilized BiConjugate Gradient)
    - AMG preconditioner
  - Presence of wall





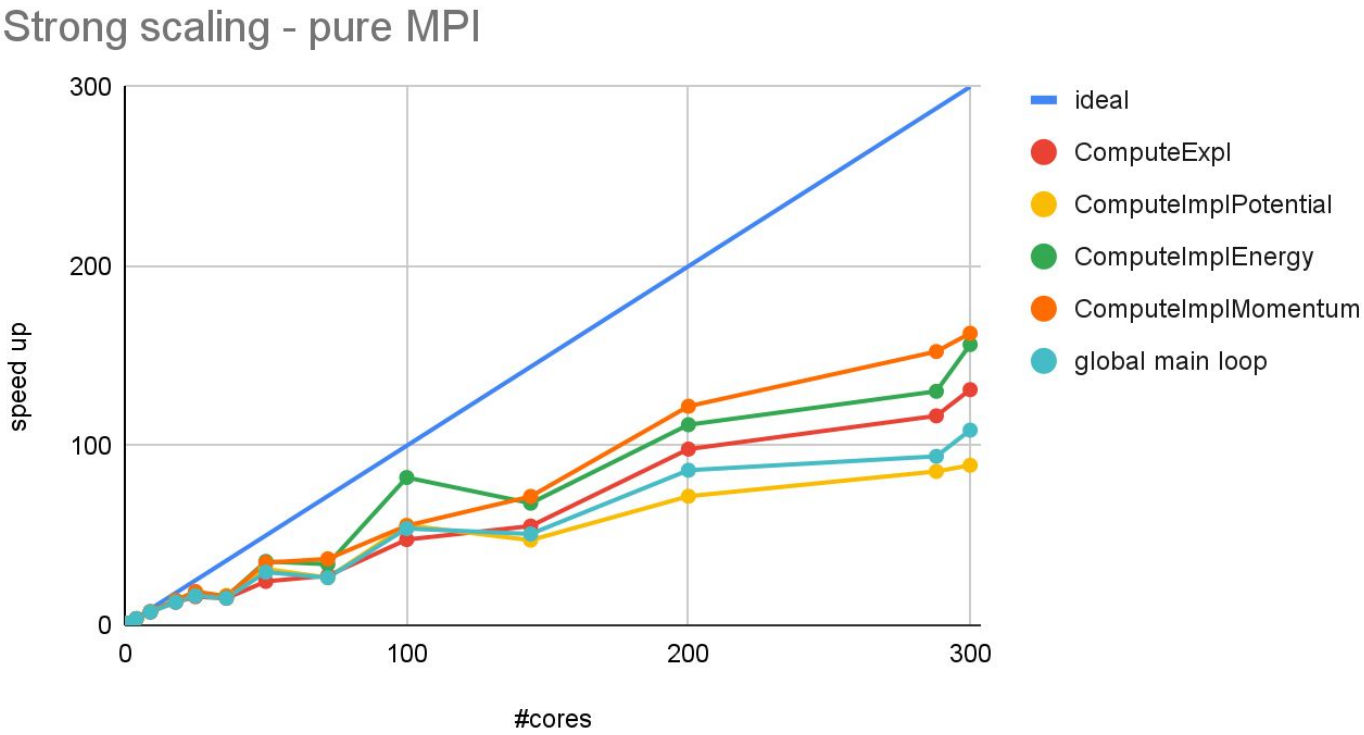
Profiling Soledge3X

- Main loop distribution for pure MPI parallelism

routines / #MPI process	ComputeExpl	ComputeImpl	ComputeImpl-ElectricPotential	ComputeImpl-Energy	ComputeImpl-Momentum
1	16%	80%	35%	25%	19%
18	16%	74%	34%	23%	18%
36	16%	74%	34%	23%	18%
72	16%	68%	35%	19%	14%
144	15%	67%	37%	19%	14%
288	13%	70%	38%	18%	12%

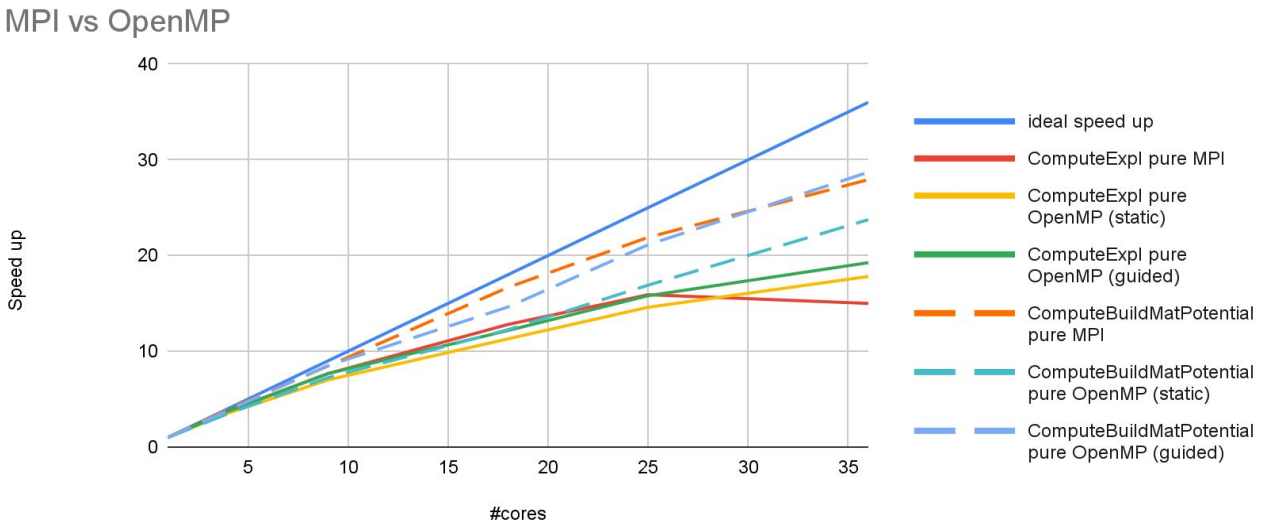
Soledge3X profiling

- Strong scaling: more efficient when the number of MPI processes divides the number of magnetic flux surfaces



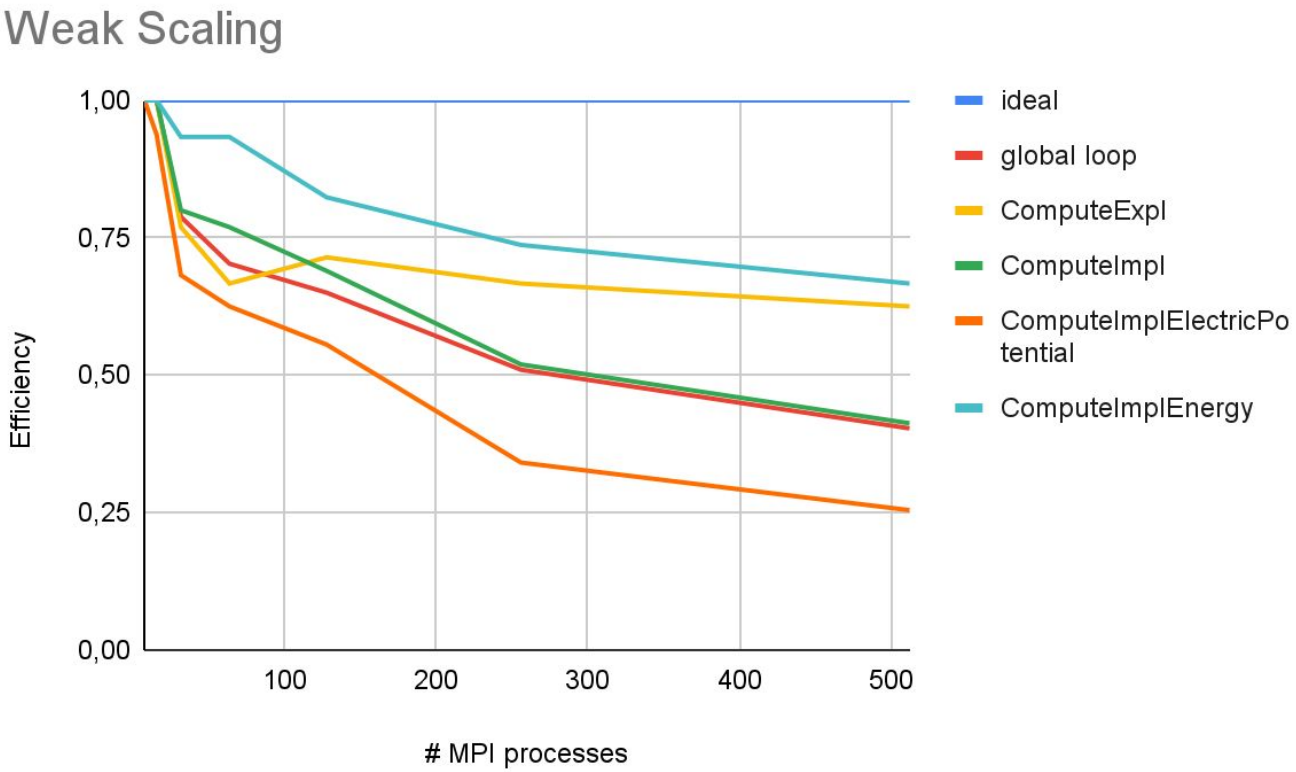
Soledge3X profiling

- Strong scaling



Soledge3X profiling

- Weak scaling



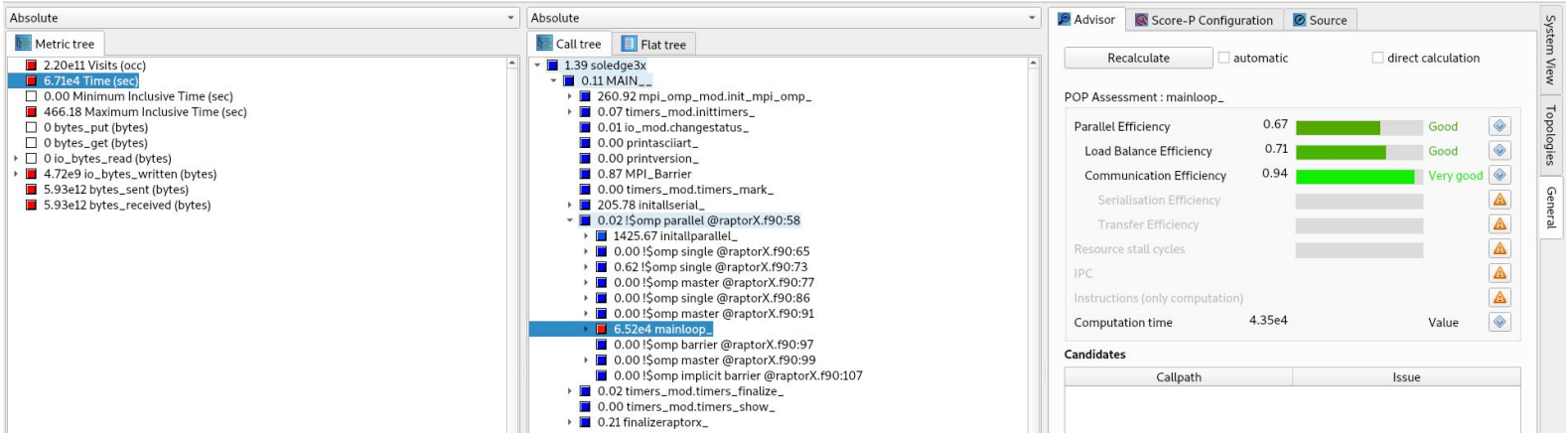
Profiling with Scorep

- Main loop: Scorep analysis for 144 MPI processes
  - Communication efficiency (*maximum across all processes of the ratio between useful computation time and total run-time*):

    CommE = maximum across processes (ComputationTime / TotalRuntime) = 0.94

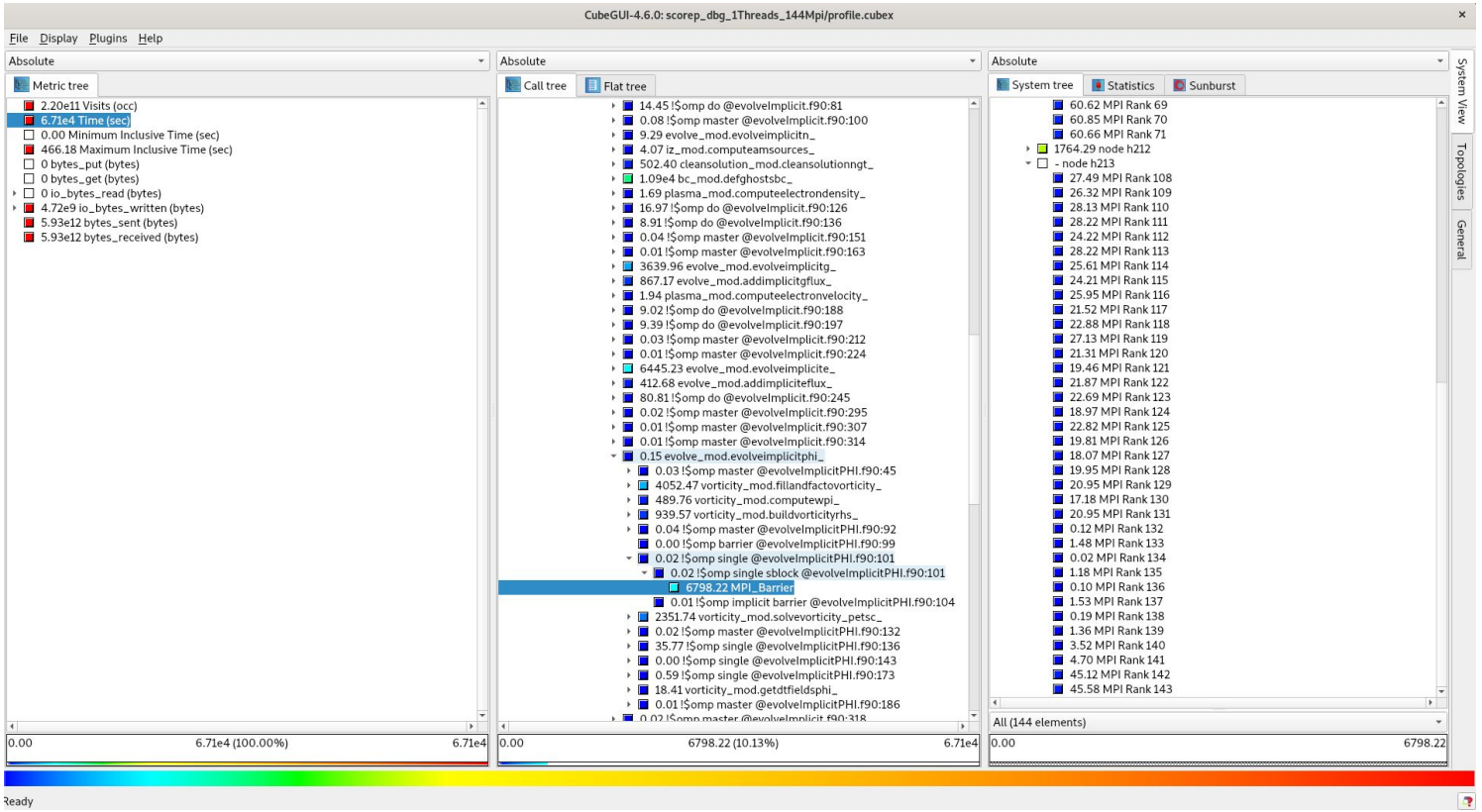
  - Load balance efficiency (*ratio between average useful computation time - across all processes - and maximum useful computation time - also across all processes -* :

    LB=avg(ComputationTime) / max(ComputationTime) = 0.71



Profiling with Scorep

- evolveImplicitPHI routine: MPI barrier take most of the time



Miniapps for linear solvers

- use of Miniapp (see Nicola's talk on solvers):
  - New routine in Soledge3X for dumping matrices in PETSC format for all implicit solvers
  - The Miniapp loads matrices and solves linear system with PETSC and AMGX (see Nicola's talk on solvers)
  - AMGX - First tests:
    - the miniapp allows the comparison between Petsc and Amgx
    - AMGX converges for matrices corresponding to 2D implicit solvers ; for matrix corresponding to the 3D Electric potential implicit solver AMGX converges only for coarse mesh
      --> need to investigate further AMGX parameters
  - HYPRE - First tests: HYPRE with OpenMP installed on Scitas cluster. Use of miniapp to test the pinning of threads to cores. Need of OpenMP nested regions to couple OpenMP threads in Soledge3X and HYPRE threads.

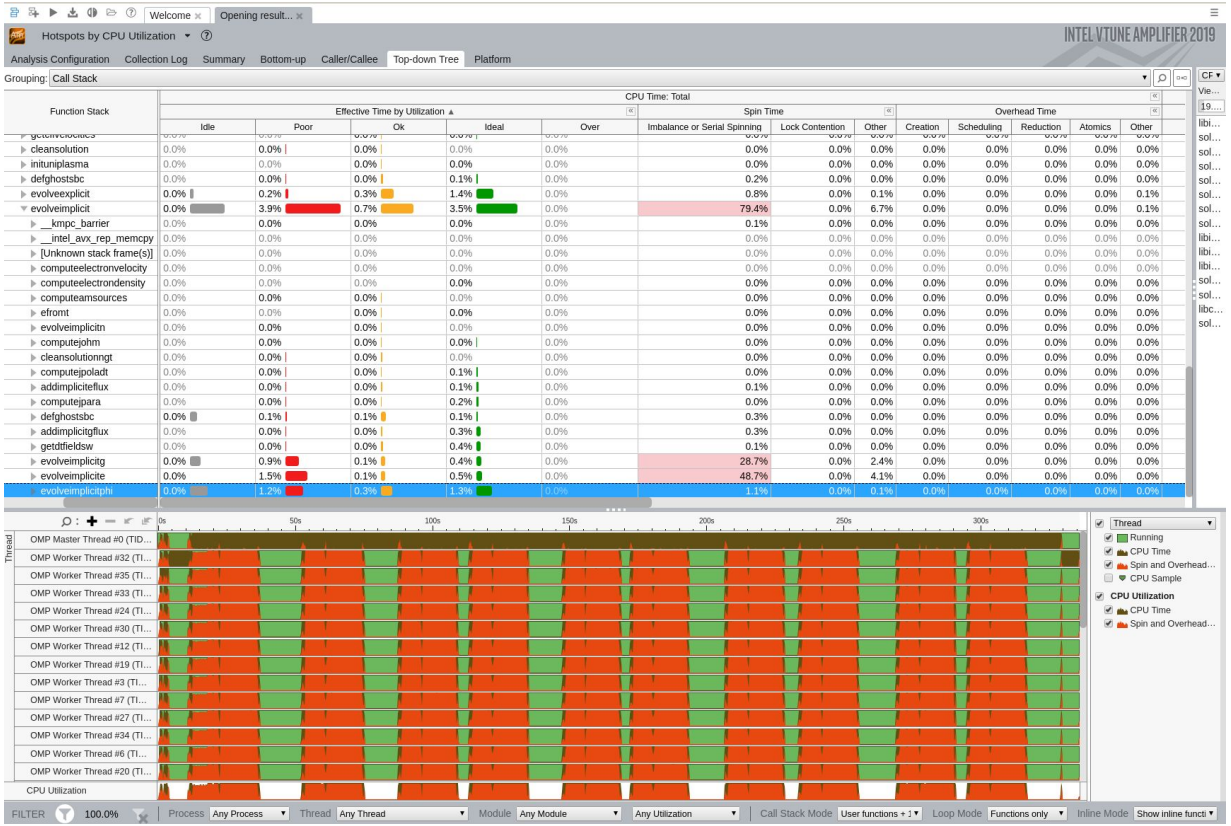
First conclusions on Soledge3X profiling

- Conclusions
  - Profiling shows most of the computation time is spent within the implicit solvers
  - MPI parallel efficiency depends on the ratio of the number of MPI processes and the number of magnetic flux surfaces
  - OpenMP is quite efficient except for linear solvers (PETSC doesn’t use threads !)

- Perspectives
  - Miniapp can help to test different linear solvers
  - Look at linear solvers using threads (Hypre ?)
  - Look at the MPI decomposition (depending in particular on the heterogeneous workload between magnetic flux surfaces and the presence of penalization mask to take into account walls)
  - Port to GPU some parts of the code
  - Overlap CPU/GPU computation
  - Intra-node optimization (OpenMP, vectorization)

intra-node profiling

- Intel-Vtune



Miniapps

- Definition: Standalone applications aimed to study specific problems.
- Usually we take the subroutines from the main codes and we turn them into standalone application.
- We need to:
  - Save the necessary data from the main codes, e.g. HDF5, NETCDF.
  - Isolate the subroutine and its dependencies(modules, libraries).
  - Create the Makefile or CMAKE.
- Advantages:
  - Easier to develop than the main codes.
  - Ideal for testing.
  - It’s doable to perform tracing.
  - Facilitate the interaction with vendors.
- Disadvantages:
  - The modification have to be injected back to the main applications.
  - The miniapp and the main application have to be maintained separately.

Miniapps relevant to the community

- Elliptic solver.
- Stencil computation.

Elliptic Solver

- The solver/preconditioner used for the Poisson equation is one of the most critical bottleneck.
- Most of the solvers involved in this project are performed in the poloidal 2D plane, with the exception of the electric potential in Soledge3X.
- What are the main solver's components?
  - Matrix building.
  - RHS building.
  - Matrix solve.
- What miniapps:
  - Solver test: just perform Ax=b. A,b inputs.
    - Purpose: To compare different methods to solve the linear system.
  - Solver + matrix build: build the matrix and solver the system.
    - Purpose: To mimic what is done in the main codes.

Solver test

- For this miniapp we read from file the matrix and rhs.
- PETSc support for now

```
call PetscViewerBinaryOpen(MPI_COMM_WORLD,mat0_file,FILE_MODE_READ,fd,ierr);
call MatCreate(MPI_COMM_WORLD,A,ierr)
call MatSetFromOptions(A, ierr)
call MatLoad(A,fd,ierr)
call PetscViewerDestroy(fd,ierr)

call PetscViewerBinaryOpen(MPI_COMM_WORLD,rhs0_file,FILE_MODE_READ,fd,ierr);
call VecCreate(MPI_COMM_WORLD,rhs_petsc,ierr)
call VecSetFromOptions(rhs_petsc, ierr)
call VecLoad(rhs_petsc,fd,ierr)
call PetscViewerDestroy(fd,ierr)
```

MAT READ

RHS READ

```
call KSPCreate(PETSC_COMM_WORLD,ksp,ierr)
call KSPSetFromOptions(ksp,ierr)
call KSPSetOperators(ksp,A,A,ierr)
call KSPSolve(ksp,rhs_petsc,lhs_petsc,ierr)
call VecNorm(lhs_petsc,NORM_2,petsc_norm,ierr)
if(rank.eq.0) write(*,*) 'PETSc norm', petsc_norm
```

SOLVE

The MatSetFromOptions, VecSetFromOptions, KSPSetFromOptions allow a great degree of flexibility

The user specify a petscrc file

PETSCRC

```
poisson_ksp_type dgmres
poisson_pc_type hypre
poisson_pc_hypre_type boomeramg
poisson_ksp_rtol 1e-7
poisson_ksp_atol 1e-15
poisson_ksp_reuse_preconditioner yes
poisson_ksp_initial_guess_nonzero yes
poisson_pc_hypre_boomeramg_strong_threshold 0.25
poisson_pc_hypre_boomeramg_max_levels 30
poisson_pc_hypre_boomeramg_coarsen_type Falgout
poisson_pc_hypre_boomeramg_agg_nl 1
poisson_pc_hypre_boomeramg_agg_num_paths 2
poisson_pc_hypre_boomeramg_truncfactor 0.2
poisson_pc_hypre_boomeramg_interp_type ext+i
```

Solver test miniapp - Grillix testcase

```python
from netCDF4 import Dataset
from petsc4py import PETSc
file = Dataset('tcv_h2.2E-2_v2.nc')
row = file['hcsr_i'][:]
col = file['hcsr_j'][:]
val = file['hcsr_val'][:]
rhs = file['rhs'][:]
npsize = file.dimensions['np'].size
A = PETSc.Mat().createAIJ(size=(npsize,npsize),nnz=file.dimensions['hcsr_nnz_dim'].size,csr=(row.data-1,col.data-1,val.data))
ksp = PETSc.KSP().create()
ksp.setOperators(A)
b = PETSc.Vec().createSeq(rhs.size)
x = PETSc.Vec().createSeq(rhs.size)
b.setArray(rhs.data)
ksp.solve(b,x)
print("Number of iterations %s norm %s" % (ksp.getIterationNumber(),x.norm()))
```

- Data in CSR.
- Input: matrix and rhs.
- Information extraction fron netcdf.
- Integration with PETSc in Python.
- Fast prototyping.

Solver test - parameter scan

```
hyperparam_values = list(itertools.product(*[json_data[d] for d in key_list]))

hyperparam = []
for h in hyperparam_values:
   hyperparam.append(dict(zip(key_list, h)))
print(len(hyperparam))

calculations = {}
for k,v in hyperparam[0].items():
   calculations[k] = []


for h in hyperparam:
   string = ''
   output_string = ''
   current_calculation = {}
   for k,v in h.items():
       string += k+' '+v+"\n"
       calculations[k].append(v)
       current_calculation[k] = v

   output_string += h['-ksp_type']+'_'+h['-pc_type']

   shutil.rmtree(output_string, ignore_errors=True)
   mkdir_p(output_string)
   f = open(output_string+'/petscrc', "w")
   f.write(string)
   f.close()
   print(string)

   with open(output_string+'/script.sh','w') as fh:


   #with open(output_string, "a") as write_file:
   # json.dump(current_calculation, write_file)
   os.system("cp fort.* %s" % output_string)
   os.chdir(output_string)
   os.system("sbatch script.sh")
   os.chdir('../')
```

1

2

- 1) Explore the configurations by generate the possible permutations solver/preconditioner specified in the hyperparameters.json
- 2) Loop over the configurations, generate the slurm script and submit to the queue.

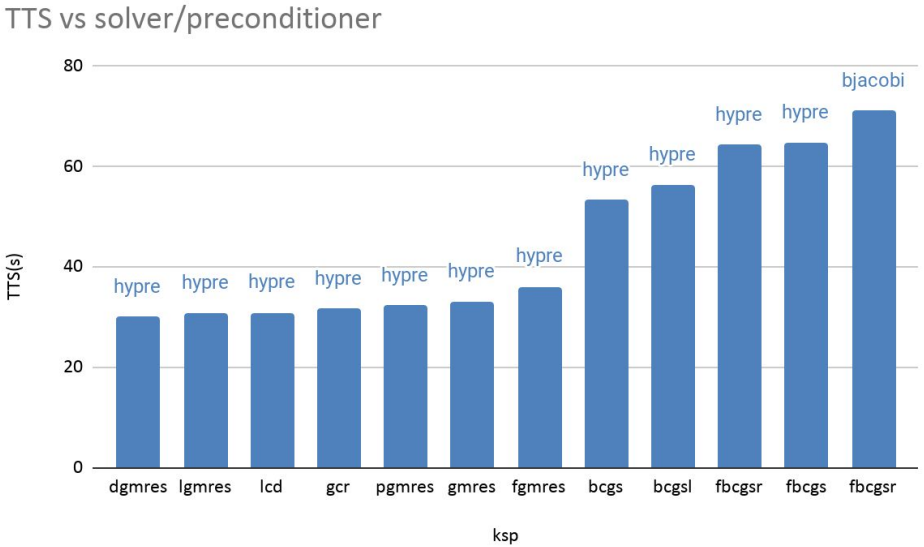
```
{
  "-ksp_rtol": ["1e-8"],
  "-ksp_type": ["bcgs", "ibcgs", "richardson", "chebyshev", "lsqr", "tfqmr"],
  "-pc_type": ["jacobi", "pbjacobi", "bjacobi", "asm",  "mg", "hypre"],
  "-ksp_inital_guess_nonzero": ["yes"]
}
```

hyperparameters.json

Solvers/Preconditioner scan

We did a scan of the possible permutations of solver/preconditioner in order to find the best performance for 1 poloidal plane

- Hypre preconditioner gives the best TTS across many solvers
- 1655 configuration tested
- Algebraic multigrid works well for these kind of solvers
- The permutations of the hypre/BoomerAMGX preconditioner parameters would require ~2M jobs
- We froze one parameter at a time for the most significant parameters in order to find the optimal configuration



Setup: Reduced TCV at 0.9T

- Turbulent mode
- Nx = 300, ny = 600
- 1 node of TAVE, KNL
- 64 cores AVX512

Solver test - discussion

- The PETSc implementation works well but matrix and vectors have to be in the PETSc format.
- It is possible to build the matrix from other formats, e.g. CSR.
- Which format shall we use?

Matrix building and solver miniapp

- Goal: to optimize the time-to-solution of matrix building plus solver.
- Building the matrix can be an expensive operation. However, it can be optimized in many cases.
- Usually the matrix is built from stencil operations, typically memory bounded.
- Test case: GBS

The solver in GBS

- The matrix building:
  - In GBS the poloidal plane has rectangular geometry and size (Nx, Ny).
  - For each point a 9 point stencil is computed, which then populate the matrix used by the solver.
  - Depending upon the solver used the matrix is built in different ways.

- The solver:
  - Direct - MUMPS
  - Iterative CPU - PETSC
  - Iterative GPU - AMGX

Matrix Assembly - PETSc CPU

DMDACreate2d

Creates an object that will manage the communication of two-dimensional regular array data that is distributed across some processors.

Synopsis

```
#include "petscdmda.h"
PetscErrorCode  DMDACreate2d(MPI_Comm comm,DMBoundaryType bx,DMBoundaryType by,DMDAStencilType stencil_type,
                             PetscInt M,PetscInt N,PetscInt m,PetscInt n,PetscInt dof,PetscInt s,const PetscInt lx[],const PetscInt ly[],DM *da)
```

Collective

Input Parameters

comm	- MPI communicator
bx,by	- type of ghost nodes the array have. Use one of DM_BOUNDARY_NONE, DM_BOUNDARY_GHOSTED, DM_BOUNDARY_PERIODIC.
stencil_type	- stencil type. Use either DMDA_STENCIL_BOX or DMDA_STENCIL_STAR.
M,N	- global dimension in each direction of the array
m,n	- corresponding number of processors in each dimension (or PETSC_DECIDE to have calculated)
dof	- number of degrees of freedom per node
s	- stencil width
lx, ly	- arrays containing the number of nodes in each cell along the x and y coordinates, or NULL. If non-null, these must be of length as m and n, and the corresponding m and n cannot be PETSC_DECIDE. The sum of the lx[] entries must be M, and the sum of the ly[] entries must be N.

MatSetValuesStencil

Inserts or adds a block of values into a matrix. Using structured grid indexing

Synopsis

```
#include "petscmat.h"
PetscErrorCode MatSetValuesStencil(Mat mat,PetscInt m,const MatStencil idxm[],PetscInt n,const MatStencil idxn[],const PetscScalar v[],InsertMode addv)
```

Not Collective

Input Parameters

mat	- the matrix
m	- number of rows being entered
idxm	- grid coordinates (and component number when dof > 1) for matrix rows being entered
n	- number of columns being entered
idxn	- grid coordinates (and component number when dof > 1) for matrix columns being entered
v	- a logically two-dimensional array of values
addv	- either ADD_VALUES or INSERT_VALUES, where ADD_VALUES adds values to any existing entries, and INSERT_VALUES replaces existing entries with new values

Notes

- We started from the automated matrix API available in PETSc
- Advantadges:
  - Automatic management of the local/global mapping.
  - To fill the matrix the user specify the local entries.
- Disadvantages:
  - Lack of control.

Test case: JT60-SA from turbulent restart

- Setup:
  - System size: Nx=1200 Ny=2000 Nz=8
  - Machine: single socket skylake with 20 cores plus V100
  - Solver: DGMRES Preconditioner: Hypre/BoomerAMG
  - 1 GBS step
- Goal: compare the performance of AMGX(GPU) vs PETSc(CPU)
- The AMGX options can be further tuned.
- It would be possible to use AmgXWrapper but it would not improve the matrix building.

PETSC/AMGX	#MPI tasks	Matrix building(s)	Solve(s)
PETSC	20(CPU)	3.35	38.4
AMGX - native API	1(GPU)	0.13	21.2

KEY POINT: TO ACHIEVE MAXIMUM PERFORMANCES IS NECESSARY TO LOOK AT ALL THE ASPECTS

From DMDA to CSR

- AMGX uses the CSR format.
- PETSc has many options for debugging: -matview

```
row 34: (2, -5.61603e-07)  (18, 8.98564e-06)  (32, -5.61603e-07)  (33, 8.98564e-06)  (34, -3.36961e-05)  (35, 8.98564e-06)  (36, -5.61602e-07)  (50, 8.98564e-06)  (66, -5.61602e-07)
row 35: (3, -5.61602e-07)  (19, 8.98564e-06)  (33, -5.61602e-07)  (34, 8.98564e-06)  (35, -3.36961e-05)  (36, 8.98564e-06)  (37, -5.61602e-07)  (51, 8.98564e-06)  (67, -5.61602e-07)
row 36: (4, -5.61602e-07)  (20, 8.98564e-06)  (34, -5.61602e-07)  (35, 8.98564e-06)  (36, -3.36961e-05)  (37, 8.98564e-06)  (38, -5.61603e-07)  (52, 8.98564e-06)  (68, -5.61602e-07)
row 37: (5, -5.61603e-07)  (21, 8.98564e-06)  (35, -5.61603e-07)  (36, 8.98564e-06)  (37, -3.36962e-05)  (38, 8.98564e-06)  (39, -5.61603e-07)  (53, 8.98564e-06)  (69, -5.61602e-07)
row 38: (6, -5.61602e-07)  (22, 8.98564e-06)  (36, -5.61602e-07)  (37, 8.98564e-06)  (38, -3.36961e-05)  (39, 8.98564e-06)  (40, -5.61602e-07)  (54, 8.98564e-06)  (70, -5.61602e-07)
row 39: (7, -5.61603e-07)  (23, 8.98564e-06)  (37, -5.61603e-07)  (38, 8.98564e-06)  (39, -3.36962e-05)  (40, 8.98564e-06)  (41, -5.61602e-07)  (55, 8.98564e-06)  (71, -5.61602e-07)
row 40: (8, -5.61602e-07)  (24, 8.98564e-06)  (38, -5.61602e-07)  (39, 8.98564e-06)  (40, -3.36961e-05)  (41, 8.98563e-06)  (42, -5.61602e-07)  (56, 8.98564e-06)  (72, -5.61602e-07)
row 41: (9, -5.61603e-07)  (25, 8.98564e-06)  (39, -5.61602e-07)  (40, 8.98564e-06)  (41, -3.36961e-05)  (42, 8.98564e-06)  (43, -5.61603e-07)  (57, 8.98564e-06)  (73, -5.61602e-07)
row 42: (10, -5.61603e-07)  (26, 8.98564e-06)  (40, -5.61603e-07)  (41, 8.98564e-06)  (42, -3.36961e-05)  (43, 8.98564e-06)  (44, -5.61602e-07)  (58, 8.98564e-06)  (74, -5.61602e-07)
row 43: (11, -5.61602e-07)  (27, 8.98563e-06)  (41, -5.61602e-07)  (42, 8.98564e-06)  (43, -3.36961e-05)  (44, 8.98563e-06)  (45, -5.61602e-07)  (59, 8.98564e-06)  (75, -5.61602e-07)
row 44: (12, -5.61602e-07)  (28, 8.98564e-06)  (42, -5.61602e-07)  (43, 8.98564e-06)  (44, -3.36961e-05)  (45, 8.98564e-06)  (46, -5.61602e-07)  (60, 8.98564e-06)  (76, -5.61602e-07)
row 45: (13, -5.61602e-07)  (29, 8.98564e-06)  (43, -5.61602e-07)  (44, 8.98564e-06)  (45, -3.36961e-05)  (46, 8.98564e-06)  (47, -5.61602e-07)  (61, 8.98564e-06)  (77, -5.61602e-07)
```

- This tool helped to understand how to create the local/global mapping in CSR.
- To port the matrix building on GPU with CUDA we had to:
  - Create the matrix with AMGX.
  - Populate the matrix with CUDA kernel.
- ~20X faster wrt to CPU. Perhaps there is room for optimization in the CPU version.

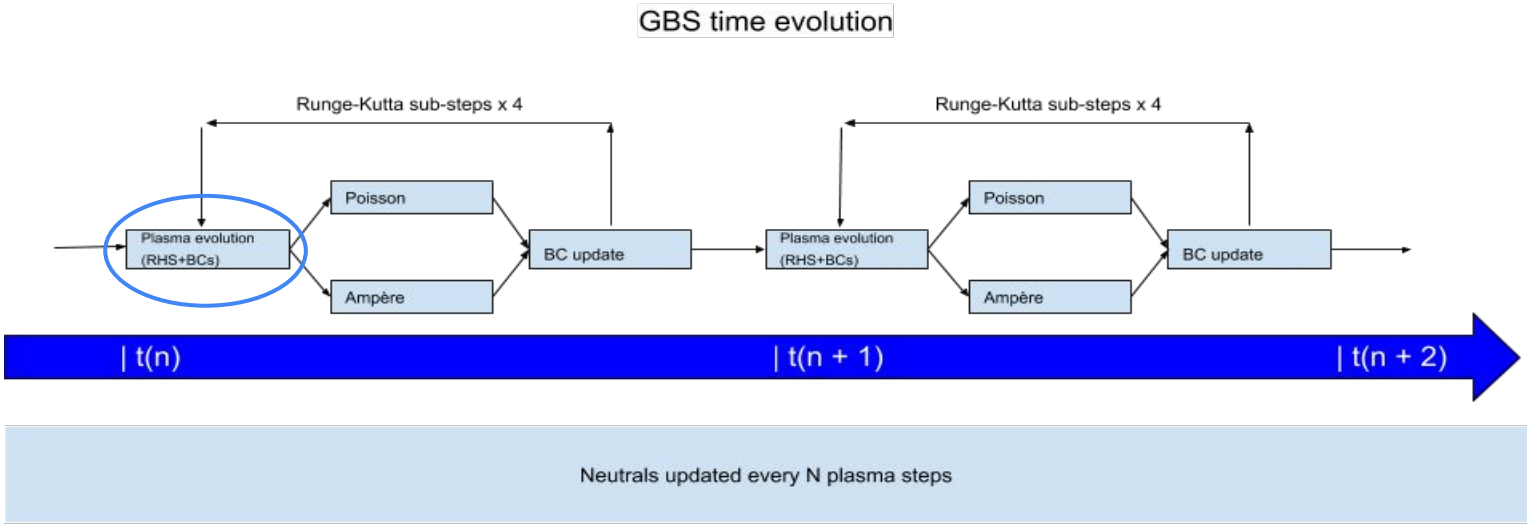
Conclusion and perspective

- Lesson 1: automatism comes at a price, **performance**.
- Lesson 2: It is necessary to choose your evil.
  - Usually, libraries and compilers support C/C++ first.
    - Sometimes there is a native fortran binding (PETSc).
    - In other cases it has to be created (AMGX).
  - In principle the matrix building could have been done with CUDA Fortran, OpenACC or OpenMP. However, only GNU C/C++ is capable to compile AMGX.
  - The hardest part of the matrix building porting was the compatibility C/Fortran.
  - However, this part can be embedded into a library, so the application developers don’t need to deal with C.
- The performance boost obtained in the matrix building is well promising for the RHS operations.

RHS computation

OpenMP Offload in GBS-RHS

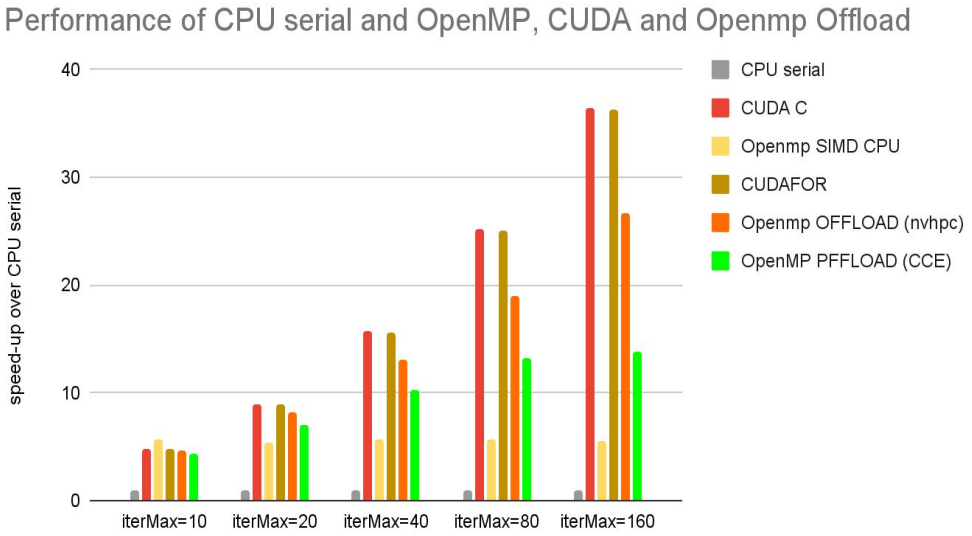
- Goal: use GPUs in plasma evolution already used in Poisson/Ampere solver
- Use of OpenMP offload for Plasma subroutines



OpenMP Offload for GPU

- CUDA only for NVIDIA GPU
- OpenMP offload for NVIDIA and AMD GPUs
  - Standardized, available for C, C++, and Fortran
  - Directive based multithreading library
  - Portable and ease of use, very good support (GNU, ARM, Intel, IBM, PGI, etc)
  - Less efficient than CUDA, high dependency on compilers
  - Performance of the classical stencil-based Jacobi example on SCITAS cluster:
    Xeon-Gold processors with 20 cores
    NVIDIA V100 GPUs (7TFLOPS)

```
! Array allocation
  allocate(A(N, N))
  allocate(Anew(N, N))
! Array initialization
  call init_array(A, N, 7.0_dp)
  call init_array(Anew, N, 7.0_dp)
! set pointers
  A_pointer1 => A
  A_pointer2 => Anew
! Jacobi iterations
  do iterk=1,iter_max
     do j = 2, N-1
        do i = 2, N-1
           A_pointer2(i, j) = 0.25 * (A_pointer1(i, j+1) + &
                A_pointer1(i, j-1) + A_pointer1(i+1, j) + A_pointer1(i-1, j))
        end do
     end do
! swap of pointers
     call swap(A_pointer1,A_pointer2)
  enddo
```



OpenMP Offload in GBS-RHS

- Example of OpenMP offloading in GBS:

```
!Compute perpendicular gradients
subroutine perpendicular_gradients
  use fields
  use array
  use gradients
  use time_integration, only: updatetlevel
  use model,only:nlpol

  use prec_const
  implicit none

  !Perpendicular gradients
  !$omp target enter data map(alloc:strmfy,strmfx,pi_y,pi_x,thetay,thetax,tempey,strmfz,pi_z)
  !$omp target enter data map(alloc:theta_curv_op,tempe_curv_op,tempi_curv_op,strmf_curv_op,theta_curv_op_v)

  !$omp task depend(in:strmf)
  call grady_n2n(strmf(:,:,:),strmfy(:,:,:))
  call gradx_n2n(strmf(:,:,:),strmfx(:,:,:))
  call gradz_n2n(strmf(:,:,:),strmfz(:,:,:))
  call curv_n2n(strmf(:,:,:), strmf_curv_op)
  !$omp end task

  !$omp task depend(in:theta)
  call grady_n2n(theta(:,:,:,updatetlevel),thetay(:,:,:))
  call gradx_n2n(theta(:,:,:,updatetlevel),thetax(:,:,:))
  call curv_n2n(theta(:,:,:,updatetlevel), theta_curv_op )
  call curv_n2v(theta(:,:,:,updatetlevel),theta_curv_op_v)
  !$omp end task

  !$omp task depend(in:tempe)
  call grady_n2n(tempe(:,:,:,updatetlevel),tempey(:,:,:))
  call curv_n2n(tempe(:,:,:,updatetlevel), tempe_curv_op)
  !$omp end task

  !$omp task depend(in:pri)
  call grady_n2n(pri(:,:,:),pi_y(:,:,:))
  call gradx_n2n(pri(:,:,:),pi_x(:,:,:))
  call gradz_n2n(pri(:,:,:),pi_z(:,:,:))
  !$omp end task
  !$omp target exit data map(from:strmfy,strmfx,pi_y,pi_x,thetay,thetax,tempey,strmfz,pi_z)

  !$omp task depend(in:tempi)
  call curv_n2n(tempi(:,:,:,updatetlevel), tempi_curv_op )
  !$omp end task

! !$omp target exit data map(from:theta_curv_op,tempe_curv_op,tempi_curv_op,strmf_curv_op,theta_curv_op_v)

end subroutine perpendicular_gradients
```

OpenMP Offload in GBS-RHS

- Example of OpenMP offloading in GBS:

```
! parallel gradient for finite differences 4rth order from n grid to n grid
subroutine gradpar_v2v_fd4(f, flu, frd, f_grad)

  use prec_const

  implicit none
  real(dp), dimension(iysg:iyeg,ixsg:ixeg,izsg:izeg), intent(in) :: f
  real(dp), dimension(iysg:iyeg,ixsg:ixeg,izsg:izeg), intent(out):: f_grad
  real(dp), dimension(iysg:iyeg,ixsg:ixeg,izsg:izeg)      :: f_z,f_y,f_x
  real(dp), dimension(iylg:ny_zg,ixsg:ixeg,2), intent(in) :: flu, frd
  integer :: ix, iy,iz

  !  f_grad(:,:,:) = nan_

  !$omp target enter data map(alloc:f_z,f_y,f_x)

  call gradz_n2n_fd4(f,f_z )
  call grady_n2n_fd4(f,f_y)
  call gradx_n2n_fd4(f,f_x)

  !$omp target teams distribute parallel do simd collapse(3)
  do iz = izs,ize
     do ix = ixs, ixe
        do iy = iys, iye
           f_grad(iy,ix,iz) = gradpar_z*f_z(iy,ix,iz) + gradpar_y_v(iy,ix)*f_y(iy,ix,iz)&
                + gradpar_x_v(iy,ix)*f_x(iy,ix,iz)
        end do
     end do
  end do
  !$omp end target teams distribute parallel do simd
  !$omp target exit data map(delete:f_z,f_y,f_x)

end subroutine gradpar_v2v_fd4
```

OpenMP Offload in GBS-RHS for GPU

- We compared initial CPU serial implementation vs OpenMP one

- Setup: Reduced TCV at 0.9T, 2 timesteps
  - Turbulent mode
  - Nx = 600, ny = 1000, nz = 4
  - 1 node piz-daint@CSCS
    - 12-core Intel Xeon 64GB RAM
    - 1 NVIDIA Tesla P100 16GB
  - Cray Compiling Environment

Subroutine	CPU serial Time(s)	OpenMP offload Time(s)	OpenMP offload Speedup
Global RHS module	**20.8**	**5.**	**4.2**
parallel gradients	11.28	1.28	9.
diffusion operators	0.5	0.25	2
interpolation	0.72	0.41	1.75

OpenMP Offload in GBS-RHS for GPU

- We compared initial CPU serial implementation vs OpenMP one

- Setup: Reduced TCV at 0.9T, 2 timesteps
  - Turbulent mode
  - Nx = 1200, ny = 2000, nz = 4
  - 1 node piz-daint@CSCS
    - 12-core Intel Xeon 64GB RAM
    - 1 NVIDIA Tesla P100 16GB
  - Cray Compiling Environment

Subroutine	CPU serial Time(s)	OpenMP offload Time(s)	OpenMP offload Speedup
Global RHS module	**80**	**16**	**5**
parallel gradients	48	3.1	15
diffusion operators	1.7	0.8	2.1
interpolation	3	1.5	2

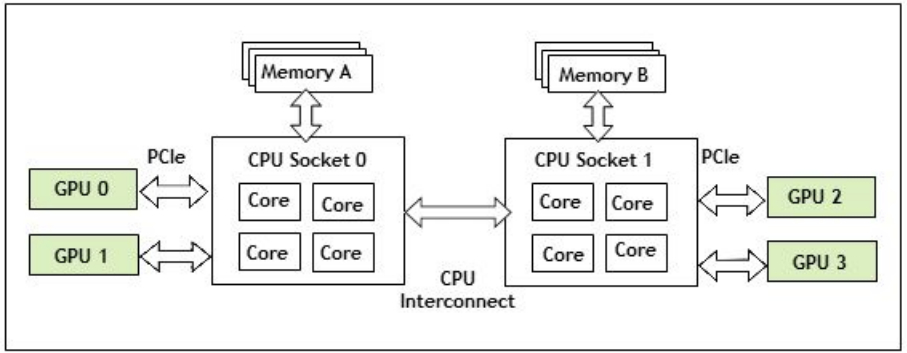
OpenMP offload in GBS

- Ongoing work:
  - asynchronous operations between CPU and GPU, identify kernels to be ported on GPU, overlap data transfer
  - To get more performance, test IBM xl compiler on MARCONI-100
  - Test new GCC 11.2 (2021-07-28) fully supporting OpenMP offload: "*For Fortran, OpenMP 4.5 is now fully supported and OpenMP 5.0 support has been extended, including the following features which were before only available in C and C++*"
  - optimize CPU/GPU data transfer

OpenMP in GBS for CPU

- Each GPU is usually associated to a single MPI process

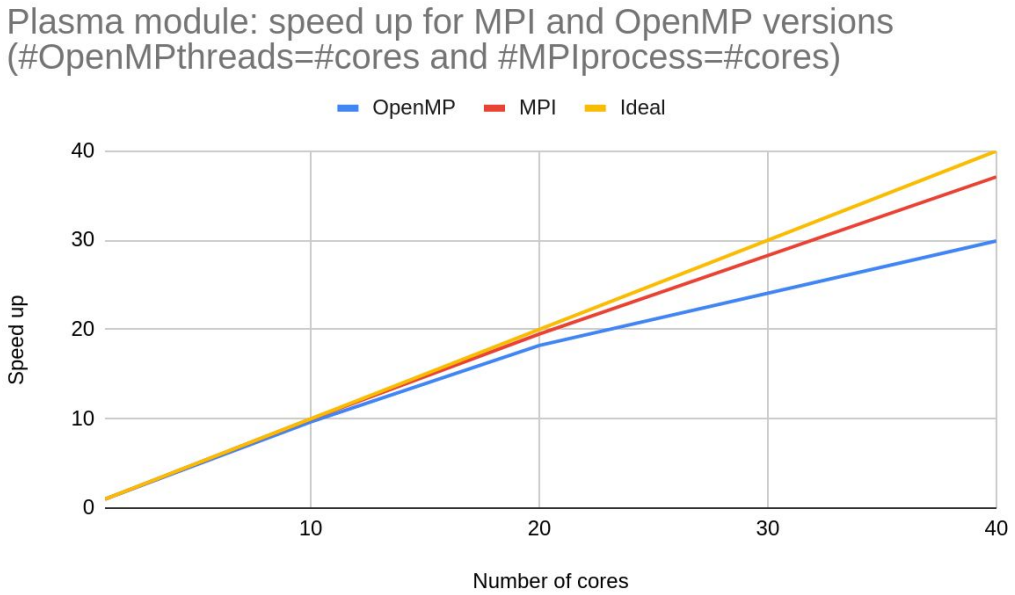
- How to exploit remaining cores :
  - associate a single core to a single MPI process (usually requiring collective MPI communications to transfer data to GPU)
  - use OpenMP to exploit remaining cores
  - use OpenMP from OpenMP-offload development is straightforward (compilation option)



OpenMP in GBS-RHS for CPU

- We compared initial CPU serial implementation, pure OpenMP one and pure MPI one

- Setup: Reduced TCV at 0.9T, 2 timesteps
  - Turbulent mode
  - Nx = 600, ny = 1200, nz = 4
  - 1 node izar
    - Xeon-Gold processors running at 2.1 GHz, with 20 cores each
    - Intel compiler



OpenACC in GBS-RHS for GPU

- Goal: adapt the work done with OpenMP offload to use OpenACC:
  --> “replace” OpenMP directives by OpenACC directives for loop and data transfer:
    - first, using unified memory with OpenACC compiling with -acc -ta=tesla:managed
    - then optimize data transfer following current openmp offload data transfer

OpenACC in GBS

- Piz daint with PGI compiler
- bandwidth (Saxpy openacc test) = 474 GB/s
- theoritical peak 4,7 TFlops
- kernel compute bound if AI>9
- Kernels in RHS are memory bound
  - kernel1: AI = 7/24 --> Memory bound
  - kernel2: AI = 31/24 --> Memory bound

kernel1

```
subroutine gradz_n2n_fd4(f , f_z)

  use prec_const

  IMPLICIT none

  real(dp), DIMENSION(iysg:iyeg,ixsg:ixeg,izsg:izeg), INTENT(in) :: f
  real(dp), DIMENSION(iysg:iyeg,ixsg:ixeg,izsg:izeg), intent(out) :: f_z
  integer :: iz
  real(dp), DIMENSION(1:4) :: coef_der

  coef_der(:) = deltazi*coef_der1_n2n(:)
!  f_z(:,:,:) = nan_
  do iz=izs,ize
     f_z(:, :, iz) =    coef_der(1)*f(:, :, iz-2)  &
          + coef_der(2)*f(:, :, iz-1)  &
          + coef_der(3)*f(:, :, iz+1)  &
          + coef_der(4)*f(:, :, iz+2)
  end do


end subroutine gradz_n2n_fd4
```

kernel2

```
subroutine gradz_v2n_fd4(f,f_z_v2n)

  use prec_const

  real(dp), DIMENSION(iysg:iyeg,ixsg:ixeg,izsg:izeg), INTENT(in) :: f
  real(dp), DIMENSION(iysg:iyeg,ixsg:ixeg,izsg:izeg), intent(out) :: f_z_v2n
  integer :: iy, iz
  real(dp), DIMENSION(1:4) :: coef_der
  coef_der(:) = deltazi*coef_der1_stag(:)

!  f_z_v2n(:,:,:) = nan_
  do iy=iys,iye
     do iz=izs,ize

        f_z_v2n(iy,:,iz) = coef_int(1)*(   coef_der(1)*f(iy-1, :, iz-1)   &
             + coef_der(2)*f(iy-1, :, iz)     &
             + coef_der(3)*f(iy-1, :, iz +1)  &
             + coef_der(4)*f(iy-1, :, iz+2)  )&

             +  coef_int(2)*(   coef_der(1)*f(iy, :, iz-1)     &
             + coef_der(2)*f(iy, :, iz)       &
             + coef_der(3)*f(iy, :, iz+1)     &
             + coef_der(4)*f(iy, :, iz+2)    )&

             +  coef_int(3)*(   coef_der(1)*f(iy+1, :, iz-1)    &
             + coef_der(2)*f(iy+1, :, iz)      &
             + coef_der(3)*f(iy+1, :, iz+1)    &
             + coef_der(4)*f(iy+1, :, iz+2)   )&

             + coef_int(4)*(   coef_der(1)*f(iy+2, :, iz-1)    &
             + coef_der(2)*f(iy+2, :, iz)      &
             + coef_der(3)*f(iy+2, :, iz+1)    &
             + coef_der(4)*f(iy+2, :, iz+2)   )

     end do
  end do

end subroutine gradz_v2n_fd4
```

OpenACC in GBS-RHS for GPU

- We compared initial CPU serial implementation vs OpenACC one

- Setup: Reduced TCV at 0.9T, 2 timesteps
  - Turbulent mode
  - Nx = 600, ny = 1200, nz = 4
  - 1 node piz-daint@CSCS
    - 1 NVIDIA Tesla P100 16GB
  - PGI Compiling Environment

Subroutine	CPU serial (one core)	OpenACC (managed option)		OpenACC (data transfer optimized)	
	time(s)	time (s)	speed up	time (s)	speed up
Global RHS module	**42.6**	**24**	**x1.75**	**10.6**	**x4**
parallel gradients	30	10.5	x2.9	0.56	x53
diffusion operators	0.5	0.5	x1	0.25	x2
interpolation	0.92	1	x0.85	0.77	x1.2
transfer HtoD/DtoH + Gpu page fault		15		4.8	

OpenACC in GBS-RHS for GPU

- Conclusion:
  - Compared to cpu (one core) version, we observed a speed up x2 for the Unified Memory OpenACC version and a speed up x4 for the OpenACC version managing data transfer
  - Memory peak usage: 10GB

- In progress:
  - optimize data transfer
  - use multiple GPU
  - Test new GNU 11.2 compiler supporting OpenACC

OpenACC in GBS-RHS for GPU

- Nsys profiling
  - Multiple streams version using *async/wait* directives