HPC AND FLEUR

11.09.2019 I ULIANA ALEKSEEVA

DRIVING THE EXASCALE TRANSITION

Mitglied der Helmholtz-Gemeinschaft

www.flapw.de

HPC = HIGH PERFORMANCE COMPUTING

Laptop:

- # cores : 2
- memory : 8 GB

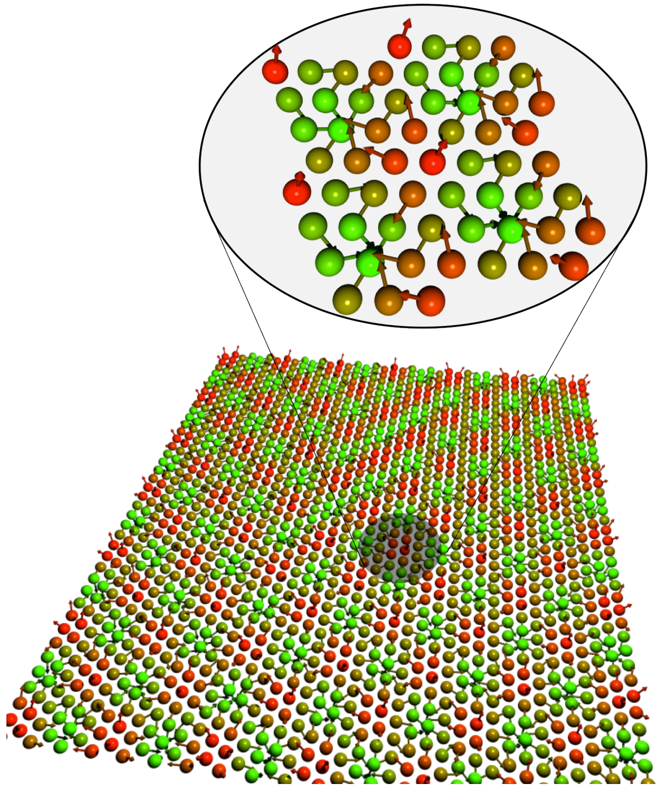
JURECA:

- # cores : 45216
- memory : 240000 GB

Source: JSC

JÜLICH Forschungszentrum

WHY BOTHER?



- Execution time ~ (Number of atoms)^3
- Memory usage ~ (Number of atoms)^2

System	# atoms	execution time	memory
NaCl	64	1 hour	2.6 GB
CuAg	256	2 days	72 GB
GaAs	512	2 weeks?	556 GB
	1000?	1 year?	
	4000?	64 years ?	

JÜLICH Forschungszentrum

OUTLINE

- HPC Hardware
- Parallel programming
- Parallelization of FLEUR
- Examples

SUPERCOMPUTER JURECA (CLUSTER MODULE)

- Top500 Jun 2019 :   #52
- 6.5 Pflops (Peta = 10^{15}, flops = floating point operations per secons)

Source: JSC

JURECA

Source: JSC

JURECA CHASSIS

Source: JSC

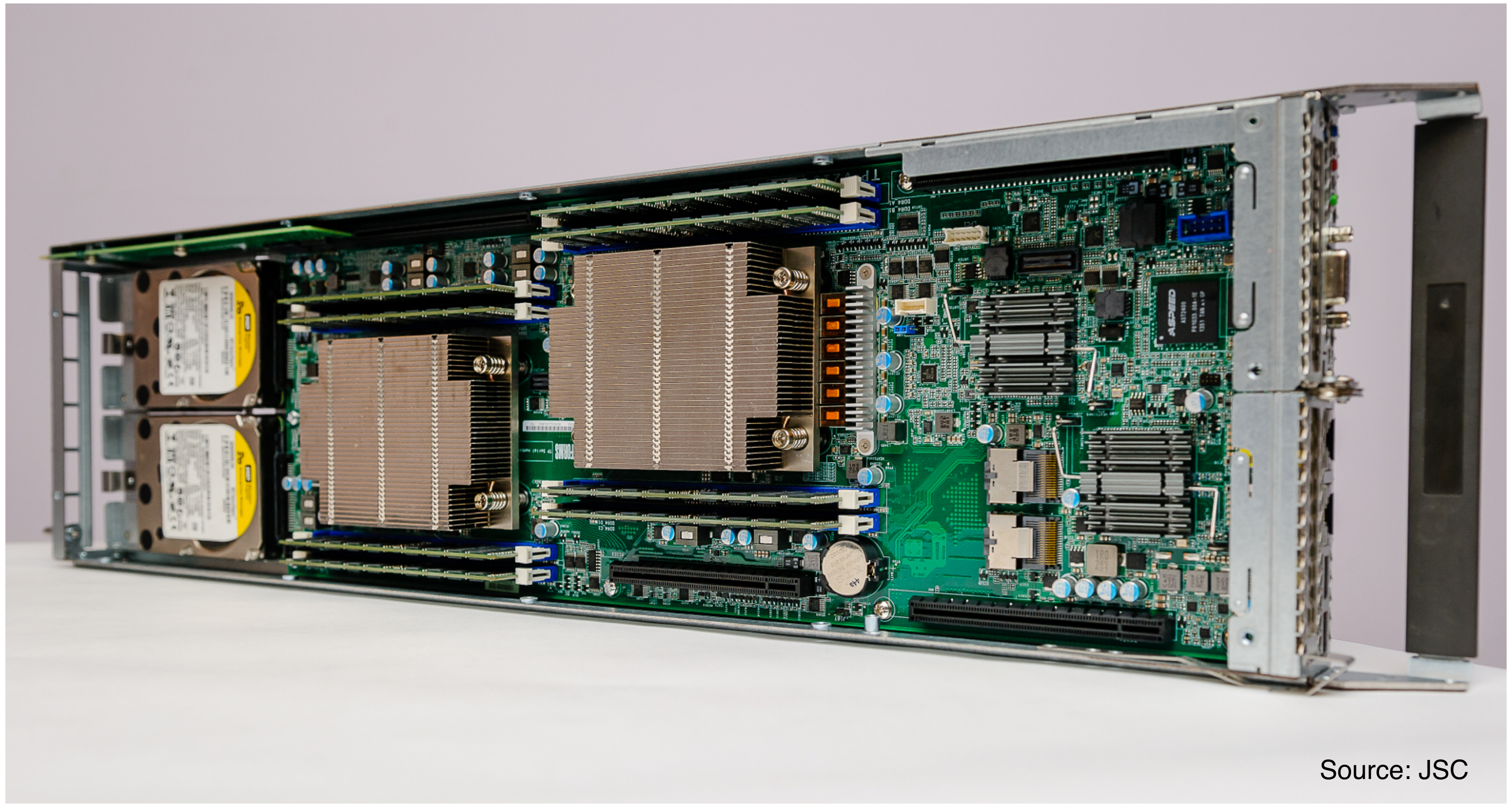
- JURECA Cluster:

  1872 compute nodes

Source: JSC

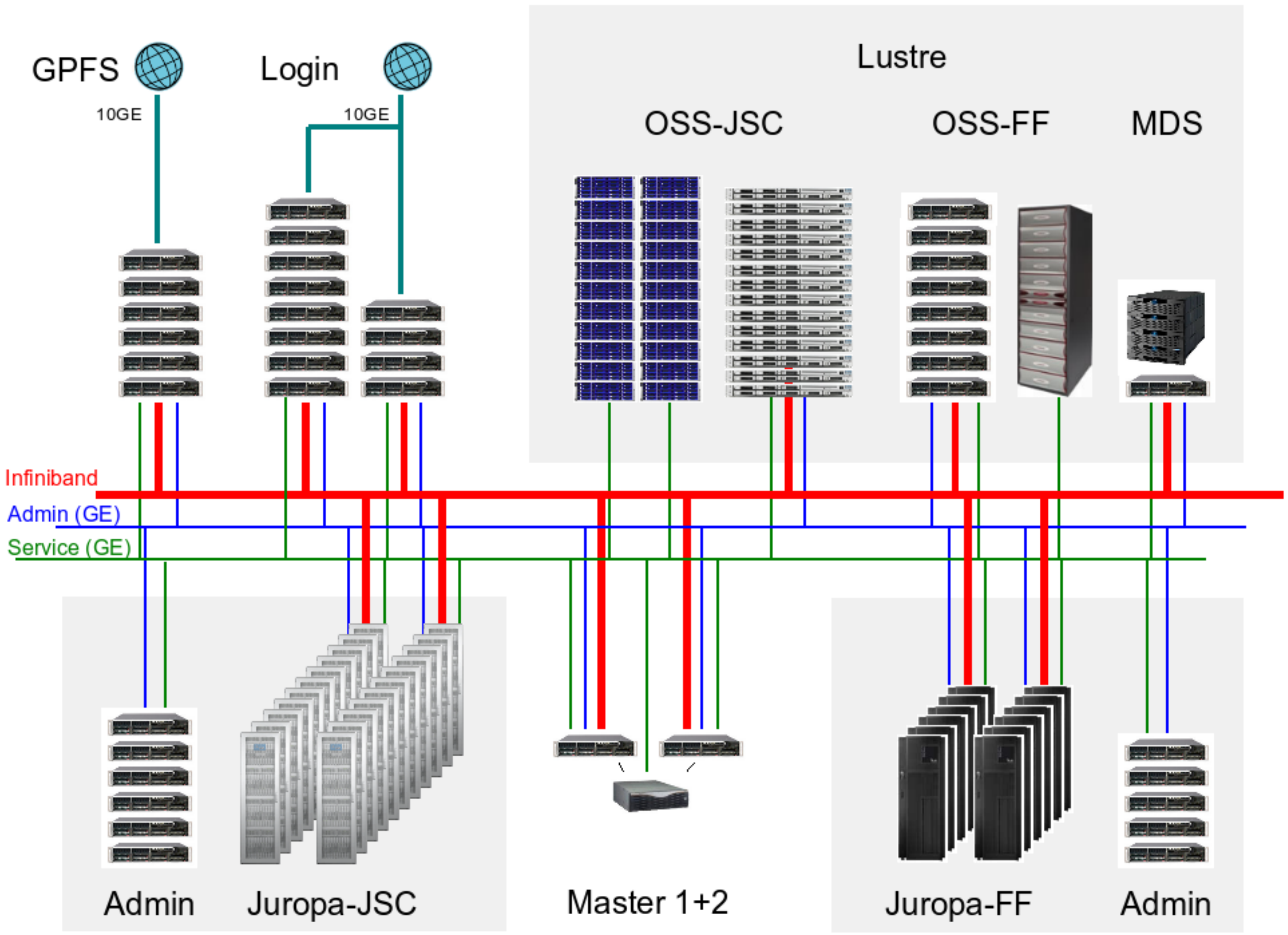
JURECA NODE

- 2 x Intel Xeon E5-2680 v3 Haswell CPUs, 12 cores each, 2.5 GHz
- 128 / 256 / 512 GiB memory
- Peak performance 960 Flops


Source: JSC

ARCHITECTURE

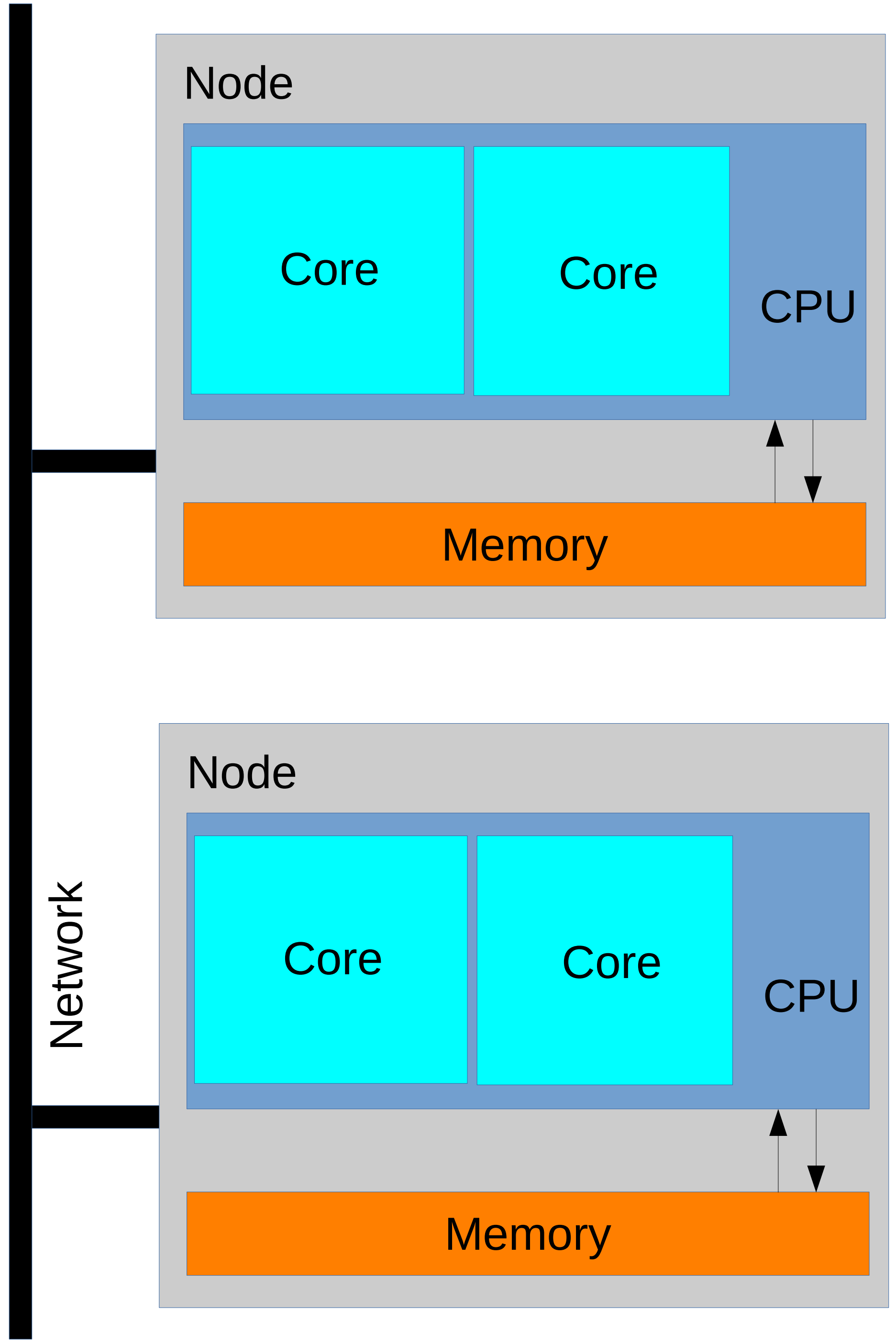
Of an HPC cluster



ARCHITECTURE

Of an HPC cluster

- cluster: many nodes
- node: compute cores + memory

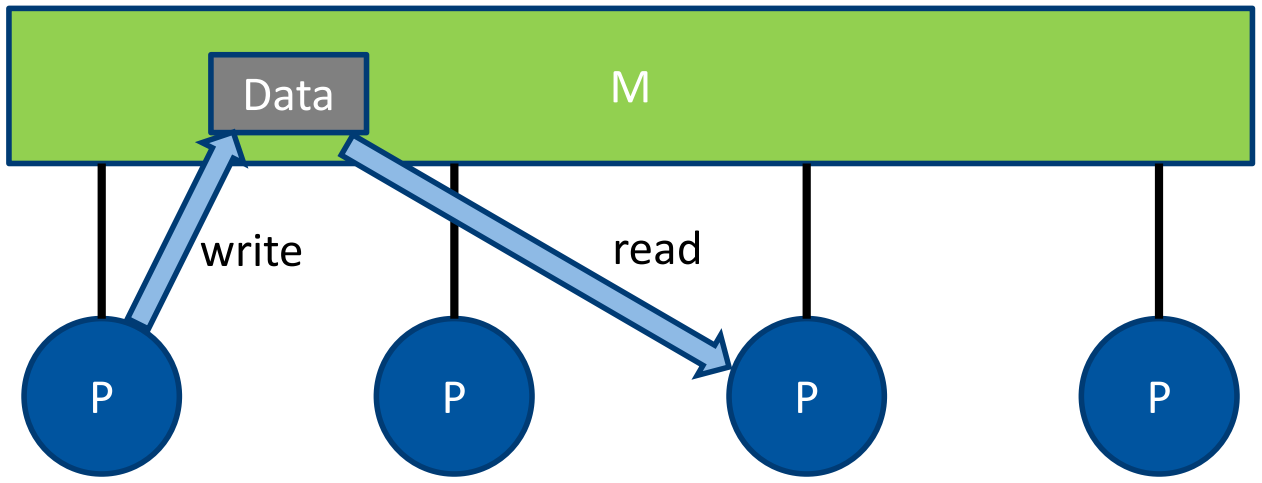


PARALLEL PROGRAMMING

Shared memory

- Memory is accessible for all processors
- No explicit communication
- Dominant standard: OpenMP

```
!---> loop over atom types
!$OMP PARALLEL DO &
!$OMP& DEFAULT(none)&
!$OMP& PRIVATE(n,nn,natom,k,i,work_r,work_c,ccchi,kspin,fk,s,r1,fj,dfj,l,df,wronk,tmk,phase,&
!$OMP& inap,nap,j,fkr,fkp,ylm,ll1,m,c_0,c_1,c_2,jatom,lmp,inv_f,lm)&
!$OMP& SHARED(noco,atoms,sym,cell,oneD,lapw,nvmax,ne,zMat,usdus,ci,iintsp,&
!$OMP& jspin,bkpt,qss1,qss2,qss3,&
!$OMP& apw,const,nobd,&
!$OMP& alo1,blo1,clo1,kvec,nbasf0,nkvec,enough,&
!$OMP& acof,bcof,ccof)
DO n = 1,atoms%ntype
   !  ----> loop over equivalent atoms
   DO nn = 1,atoms%neq(n)
      natom = 0
      DO i = 1, n-1
         natom = natom + atoms%neq(i)
      ENDDO
      natom = natom + nn
      IF ((atoms%invsat(natom).EQ.0) .OR. (atoms%invsat(natom).EQ.1)) THEN
         !--->          loop over lapws
         IF (zmat%l_real) THEN
            ALLOCATE ( work_r(nobd) )
         ELSE
            ALLOCATE ( work_c(nobd) )
         ENDIF
         DO k = 1,nvmax
            IF (.NOT.noco%l_noco) THEN
               IF (zmat%l_real) THEN
                  work_r(:ne)=zMat%z_r(k,:ne)
               ELSE
```



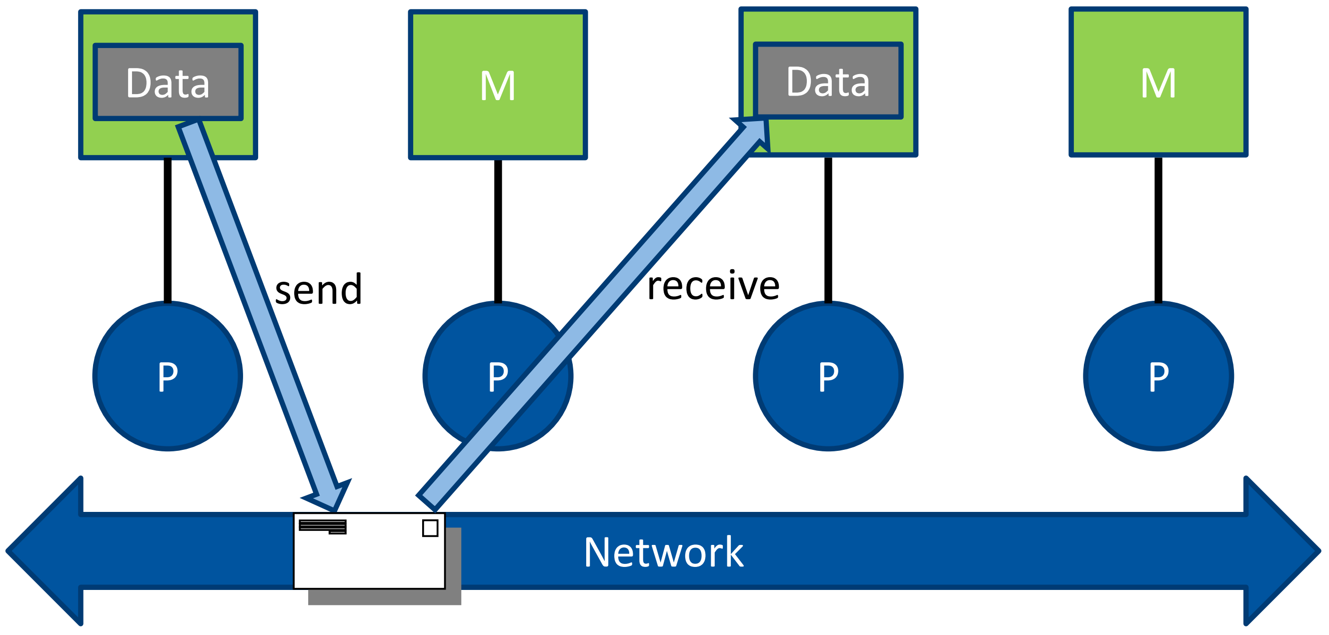
Source: H.Iliev

PARALLEL PROGRAMMING

Distributed memory

- Each processor only access its own memory
- Explicit communication between processors
- Dominant standard: MPI

```
 if(isize.ne.1)then
  do ikpt=1,fullnkpts
   if(l_p0)then
    do cpu_index=1,isize-1
     if(mod(ikpt-1,isize).eq.cpu_index)then
      call MPI_RECV(
&             matrix4(1:num_dims,1:num_bands1,
&             1:num_bands2,ikpt),
&             num_bands1*num_bands2*num_dims,
&             CPP_MPI_COMPLEX,cpu_index,
&             ikpt,mpi_comm,stt,ierr)
     endif !processors
    enddo !cpu_index
   else
    if(mod(ikpt-1,isize).eq.irank)then
      call MPI_SEND(
&             matrix4(1:num_dims,1:num_bands1,
&             1:num_bands2,ikpt),
&             num_bands1*num_bands2*num_dims,
&             CPP_MPI_COMPLEX,0,
&             ikpt,mpi_comm,ierr)
    endif !processors
   endif ! l_p0
   call MPI_BARRIER(mpi_comm,ierr)
  enddo !ikpt
 endif !isize
```

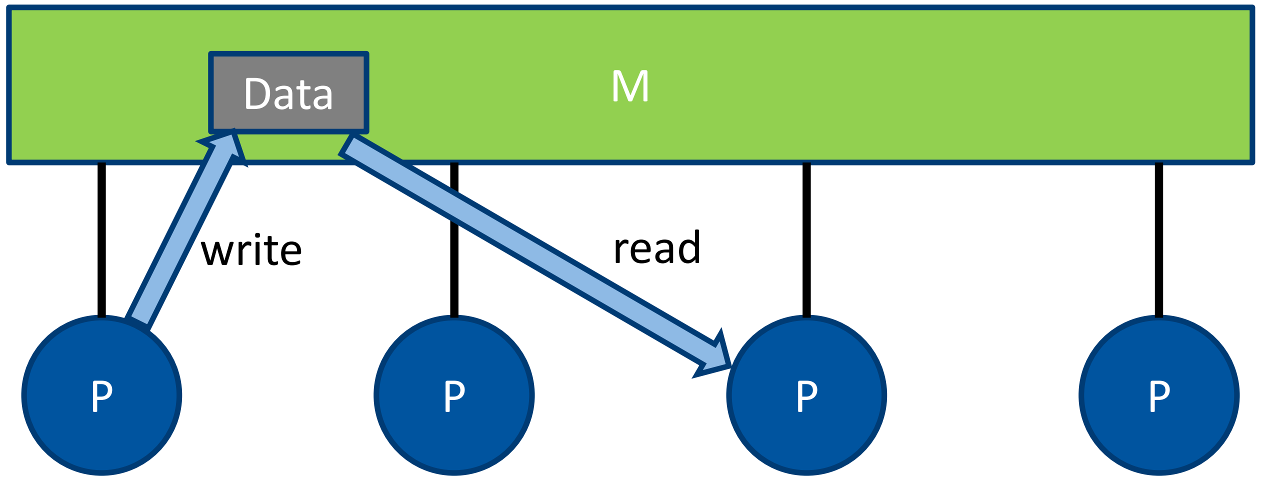


Source: H.Iliev

PARALLEL PROGRAMMING

Shared memory

- **Q:** Can I use message passing strategy inside a node?



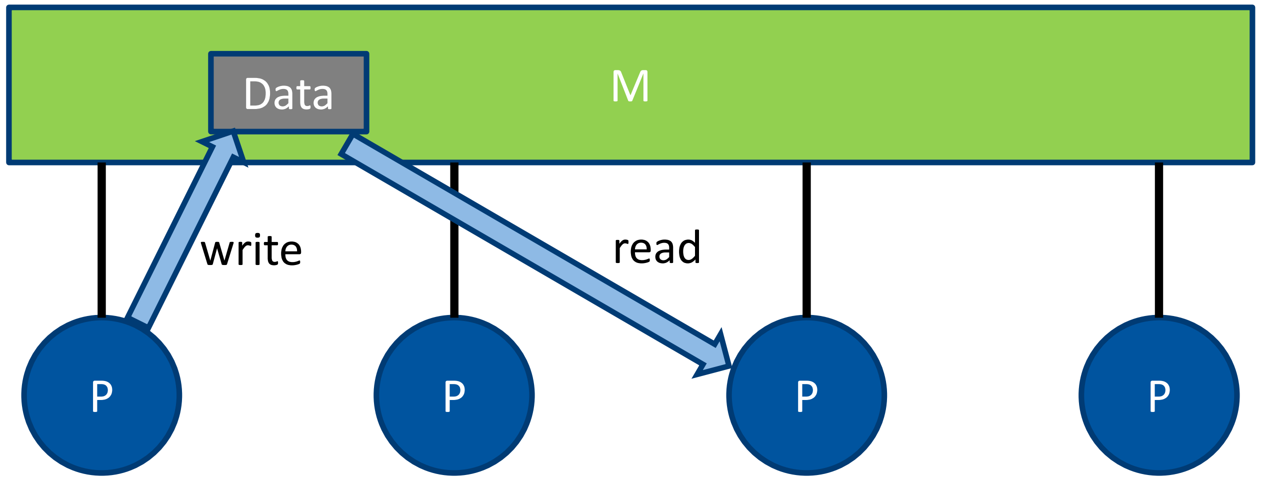
Source: H.Iliev

PARALLEL PROGRAMMING

Shared memory

- **Q:** Can I use message passing strategy inside a node?
- **A:** It's possible, but might be inefficient

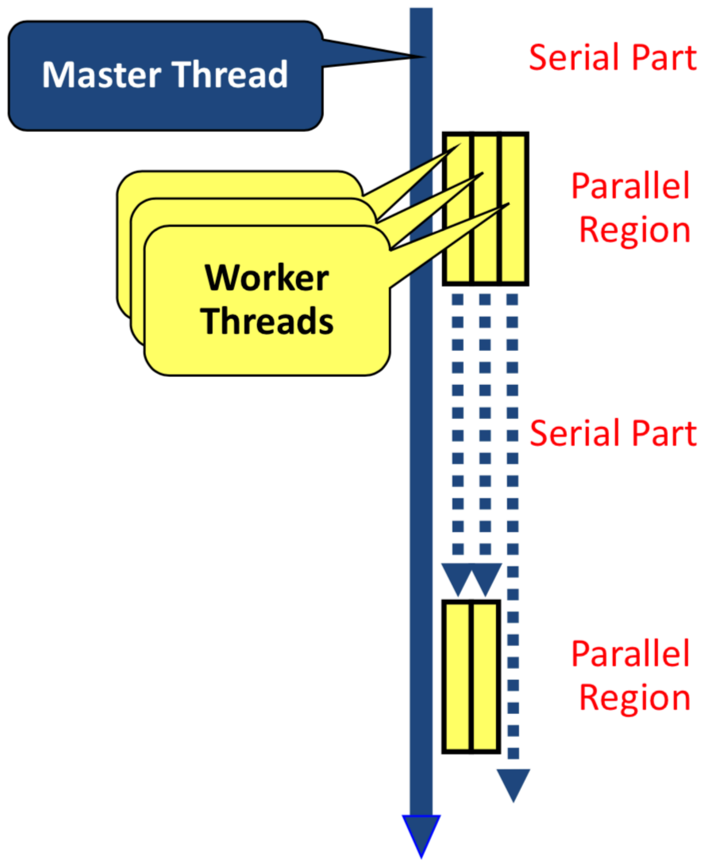
Hybrid parallelizm:
MPI + OpenMP



Source: H.Iliev

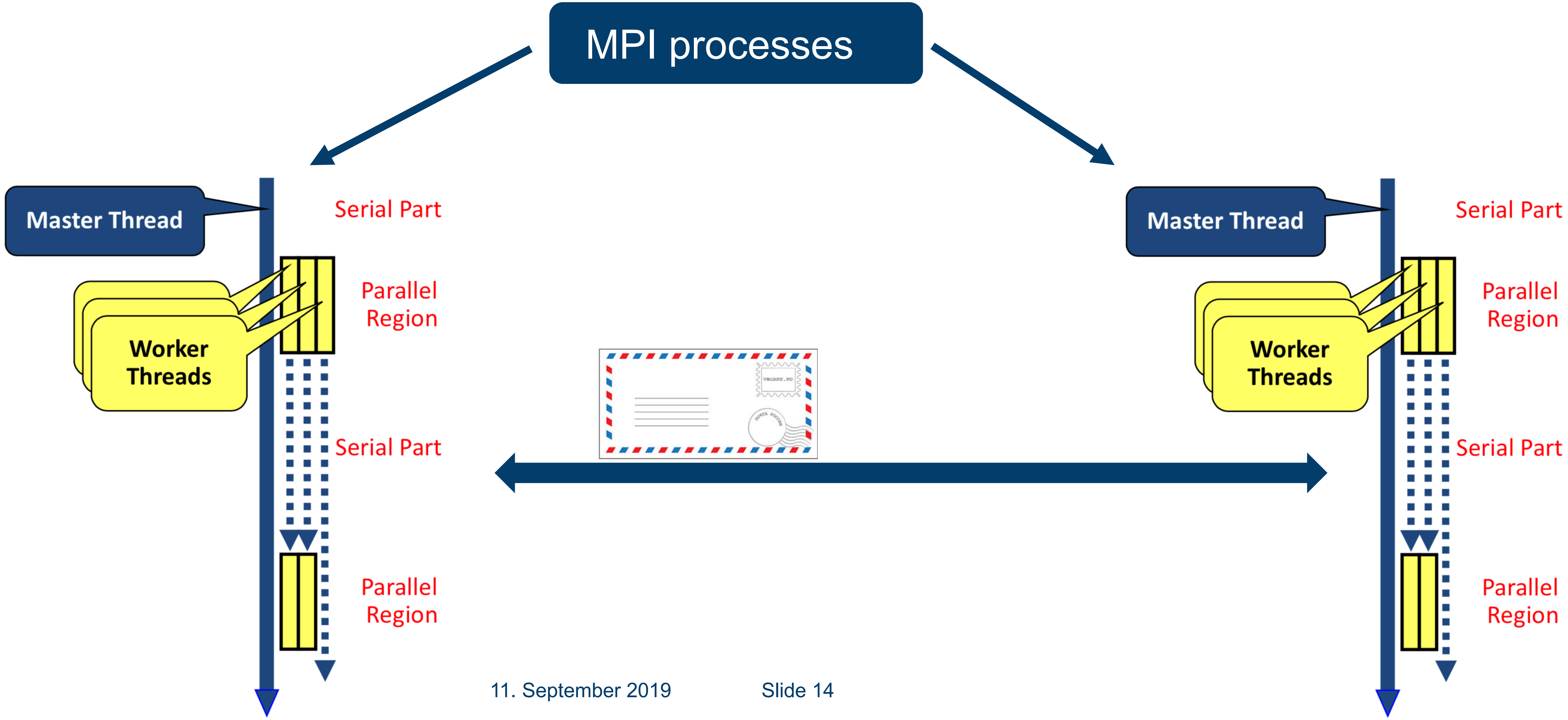
PARALLEL PROGRAMMING

Execution model: OpenMP



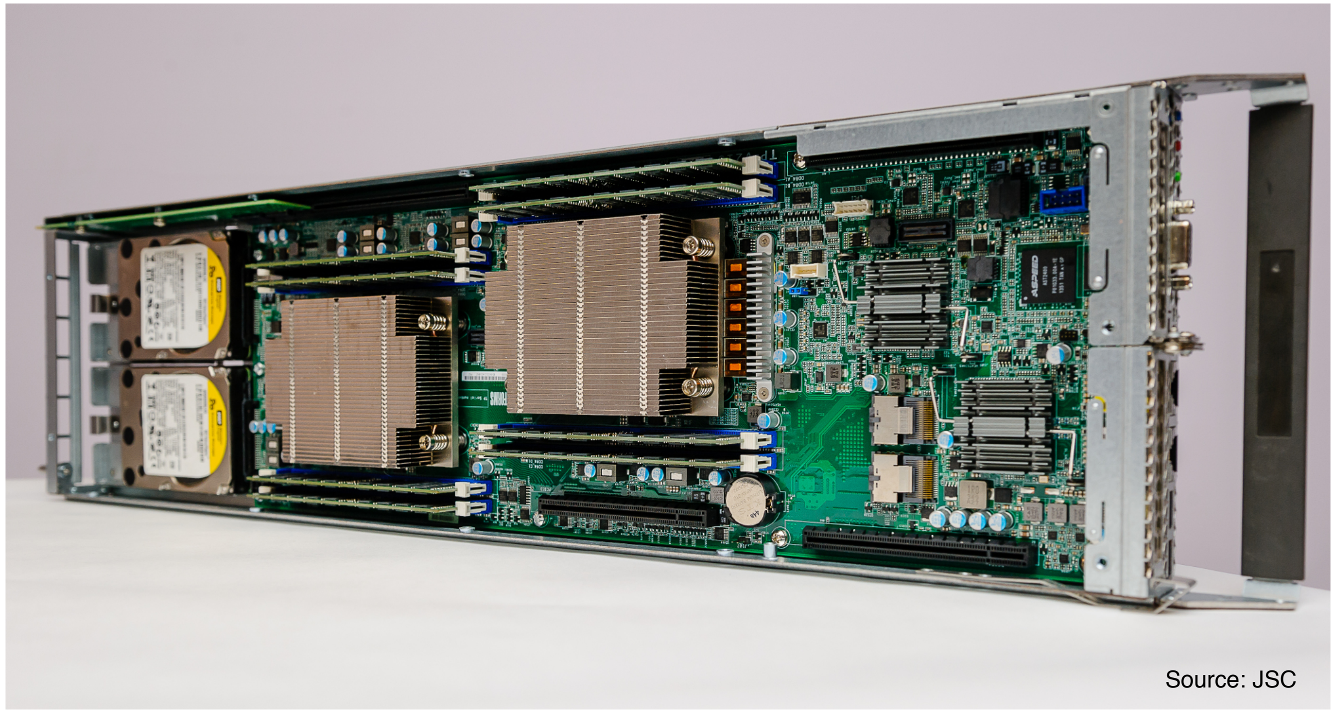
PARALLEL PROGRAMMING

Execution model: MPI + OpenMP



PARALLEL PROGRAMMING

Mapping threads onto cores



- JURECA node: 24 cores
- How to map?
  - 1 MPI x 24 threads
  - 2 MPI x 12 threads
  - 4 MPI x 6 threads
  - 12 MPI x 2 threads
  - 24 MPI
- Good starting point : 2-4 MPI processes per node

JÜLICH Forschungszentrum

FLEUR

Algorithm

Potential

$$V[n]$$

H, S- dense Hermitian matrices

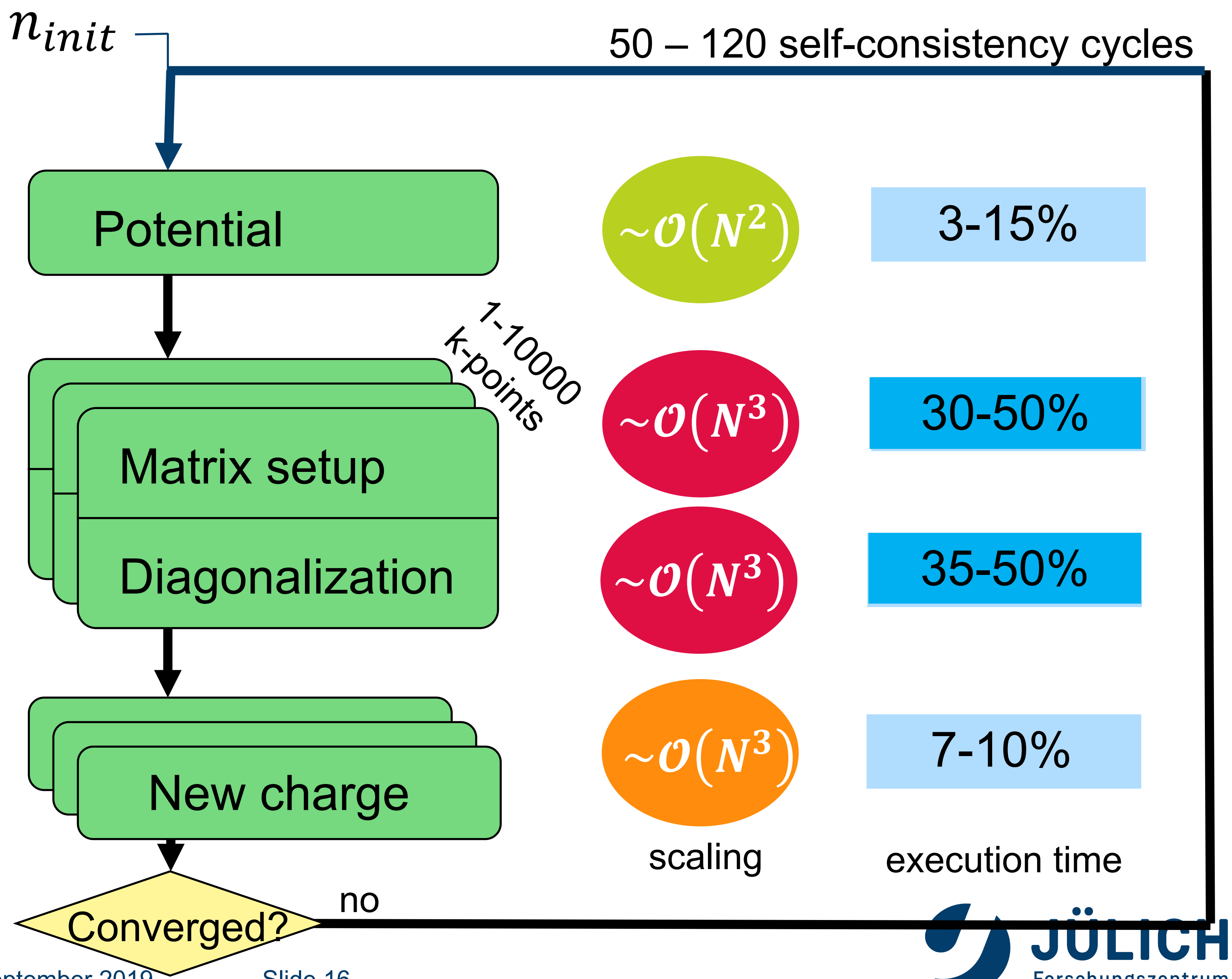
$$H = T + V[n]$$

Generalized eigenvalue problem

$$H\psi_i = \varepsilon_i S\psi_i$$

Charge density

$$n(r) = \sum_i^{occ} |\psi_i|^2$$



JÜLICH Forschungszentrum

FLEUR

Hybrid MPI + OpenMP parallelization

Three levels of parallelization:

- MPI over k-points
- MPI eigenvector parallelization
- OpenMP parallelization

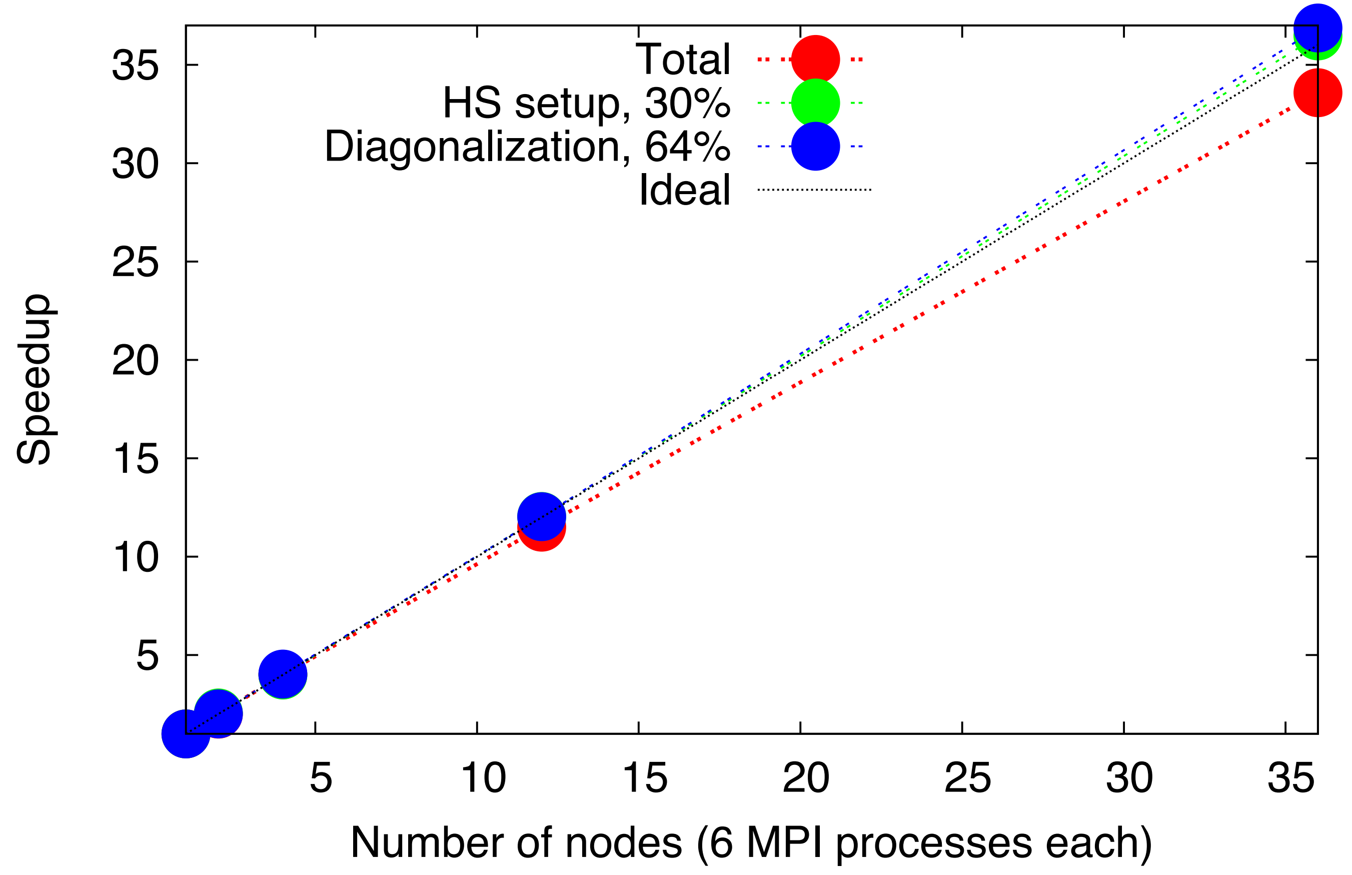
Intel Broadwell EP E5-2650v4
24 cores/node

K-POINT PARALLELIZATION

Ideal scaling

- most time-consuming part of the code are independent for different k-points
- FLEUR will distribute k-points to maximize the load balance
- try to adjust #k-points and #MPI processes

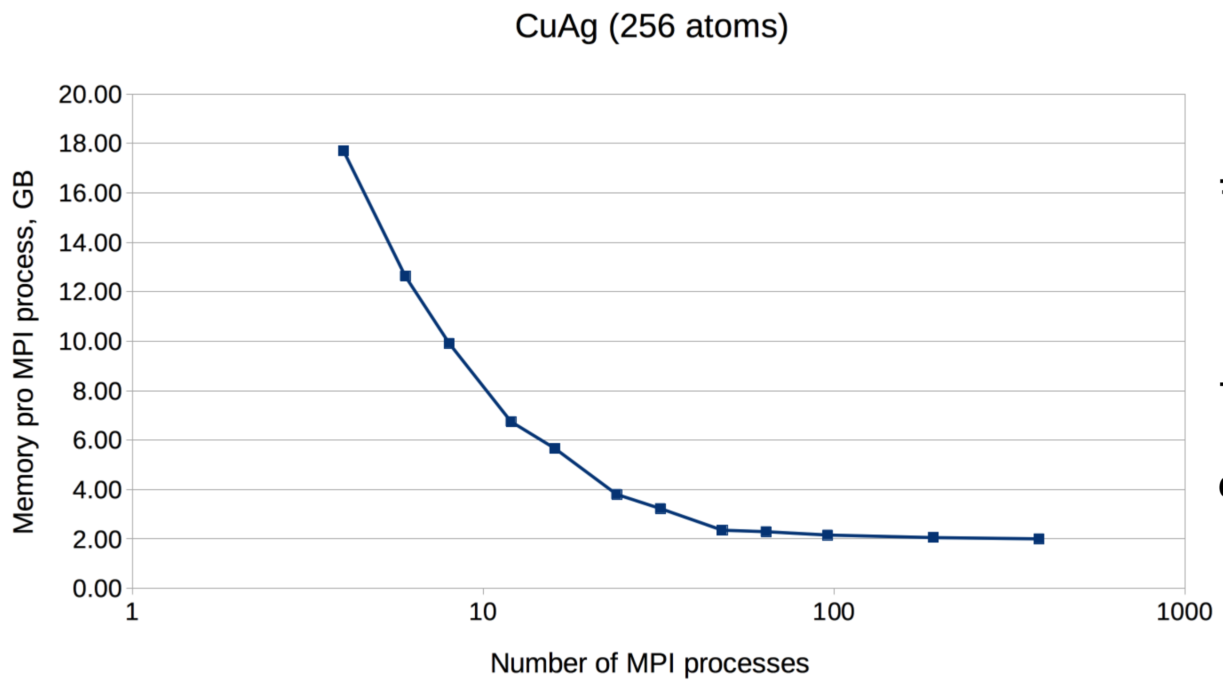
NaCl (64 atoms, 216 k-points)

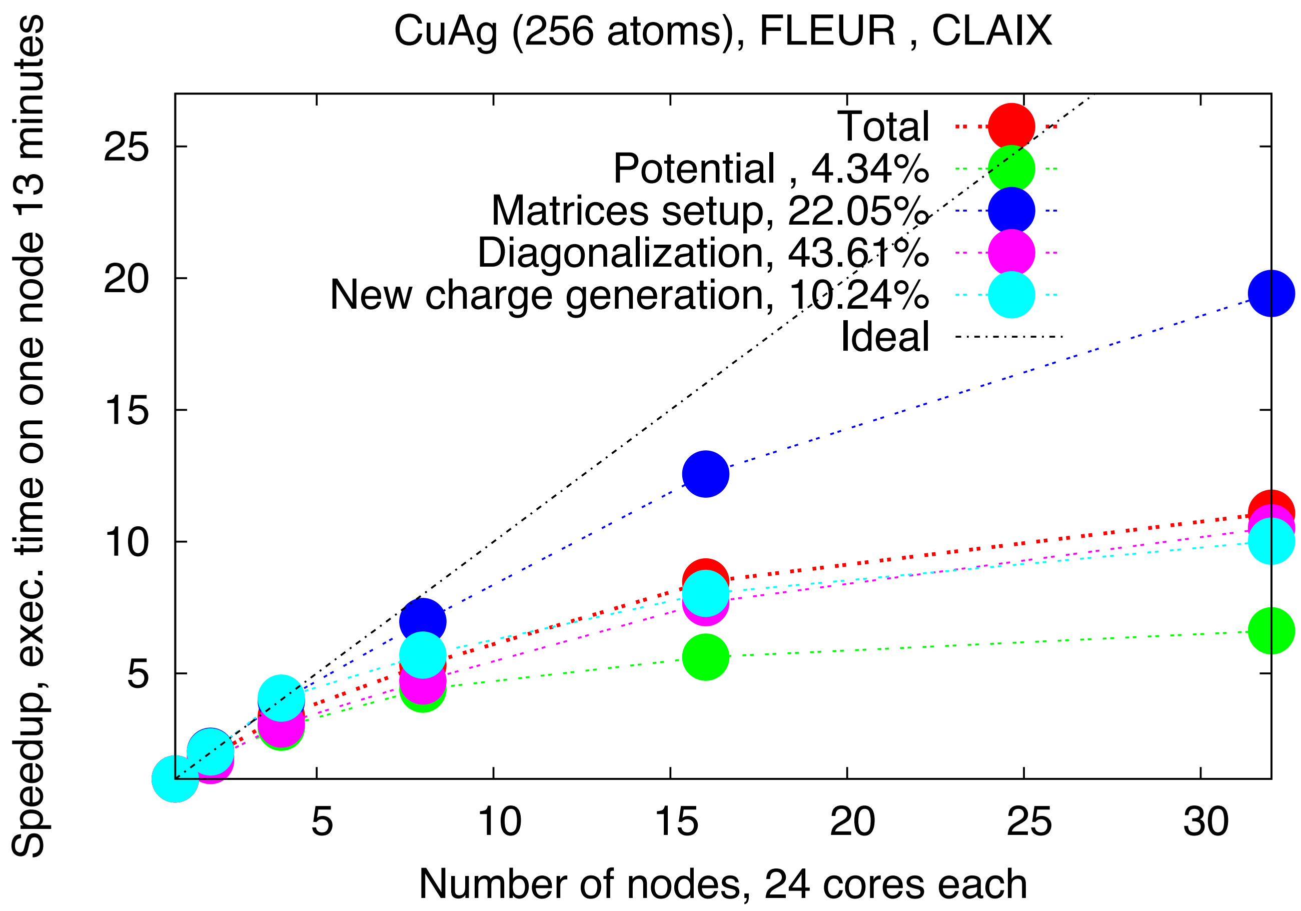


THE EIGENVECTOR PARALLELIZATION

Intel Broadwell EP E5-2650v4
24 cores/node

- gives an additional speedup
- allows to tackle larger systems by reducing the memory usage per MPI process





FLEUR IS PART OF MAX PROJECT

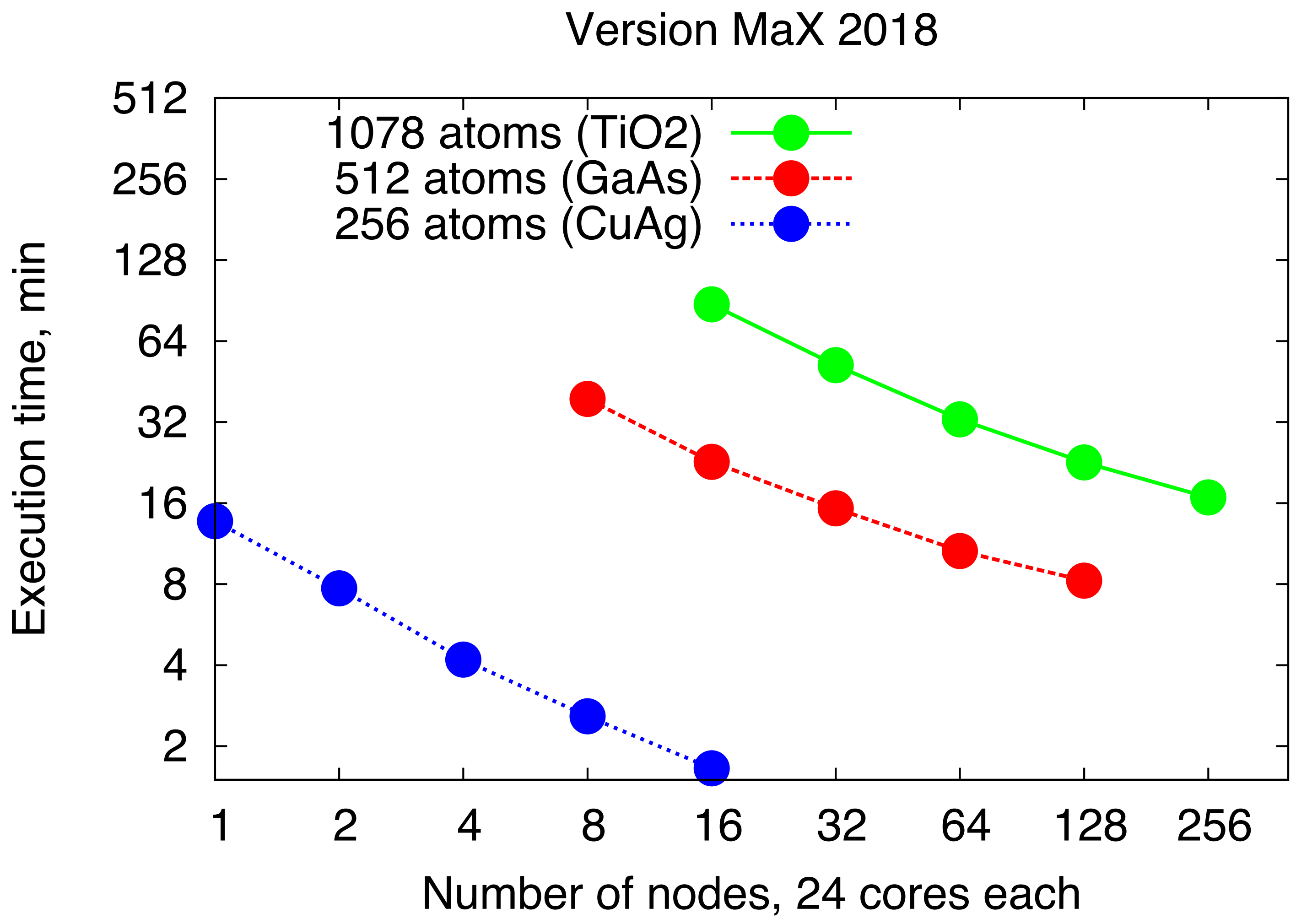
- MaX: Materials design at the exascale
- a European centre of excellence
- Top500 #1: “Summit” 200 PFlops
- Exascale:

  USA 2021 “Aurora” ,“Frotier”

  EU 2022/2023

FLEUR PERFORMANCE

Intel Broadwell EP E5-2650v4
24 cores/node



- MaX Release 3.0 (2018)
- www.flapw.de
- >1000 atoms
- larger systems scale further

New possibilities
realistic systems:
extended defects,
surfaces,
disordered solids…

U. Alekseeva et al. in Euro-Par 2018: Parallel Processing https://doi.org/10.1007/978-3-319-96983-1_52

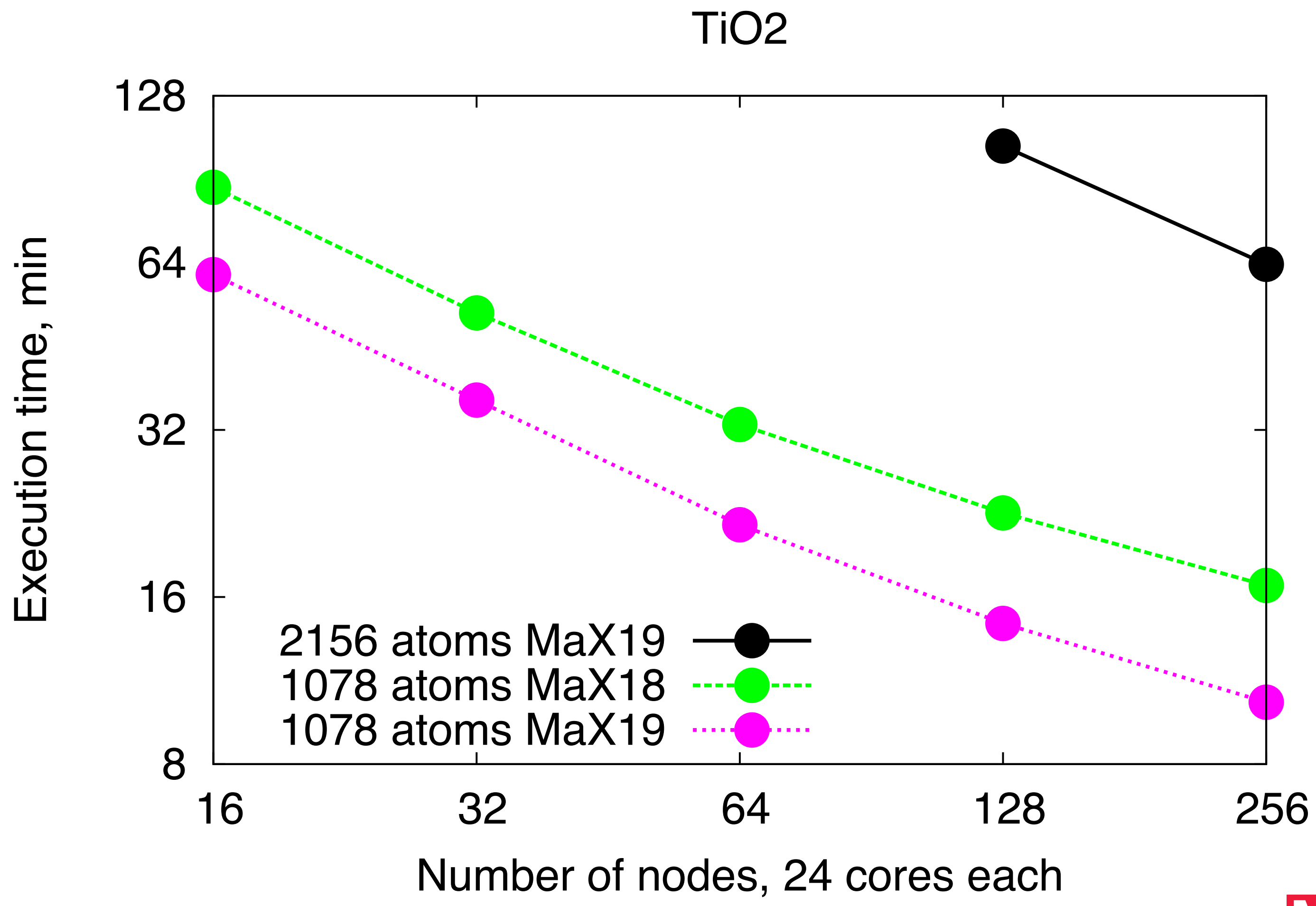
MAX DRIVING THE EXASCALE TRANSITION

JÜLICH Forschungszentrum

Intel Broadwell EP E5-2650v4
24 cores/node

FLEUR PERFORMANCE

improves with every release ;)



- MaX Release 3.0 (2018) vs.
  Max Release 3.1 (2019)
- new data storage
- >2000 atoms

MAX DRIVING THE EXASCALE TRANSITION

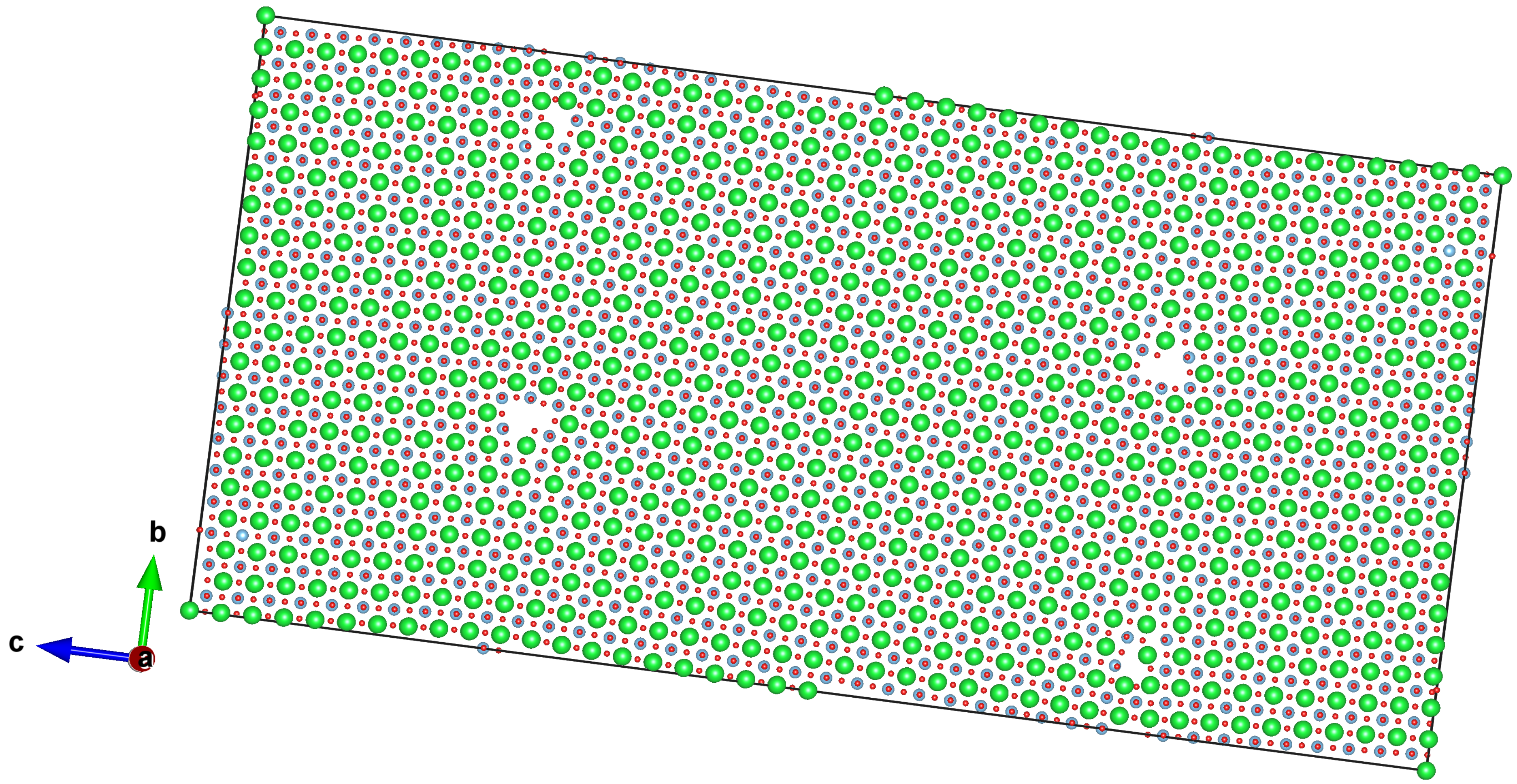
JÜLICH Forschungszentrum

DISLOCATION DEFECT

$SrTiO_3$

3750 atoms

1 iteration 1 k-point:
3h on 512 nodes
(12288 cores)

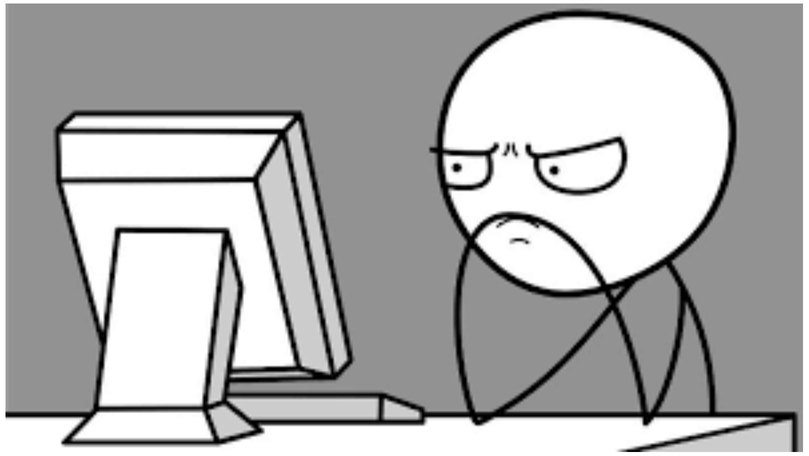


MAX DRIVING THE EXASCALE TRANSITION

JÜLICH Forschungszentrum

USING FLEUR

Good practices



- **#nodes:**

  how much memory do you need?

  don’t scale too much (<16x)

- **#MPI processes:**

  corresponds to #k-points

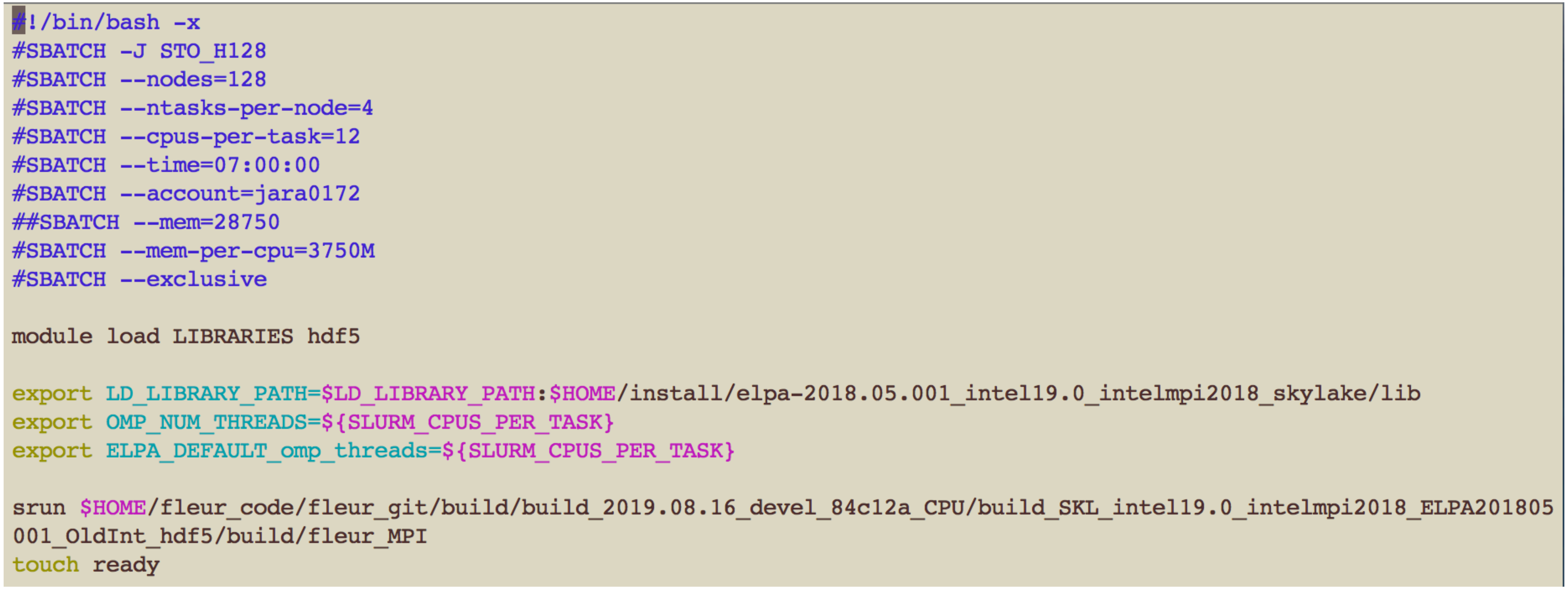
- **#threads:**

  use all cores

  1-2 MPI processes per socket

  no full hyperthreading

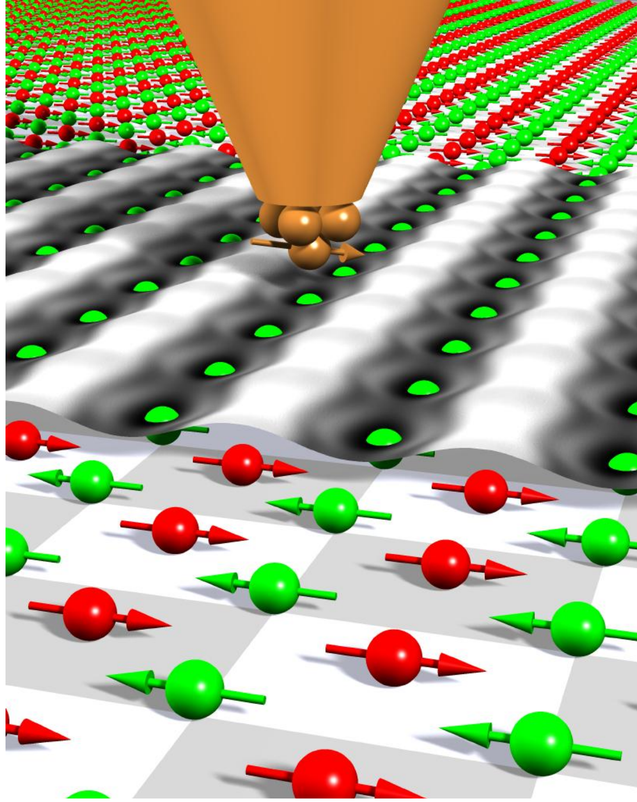
BATCH FILE

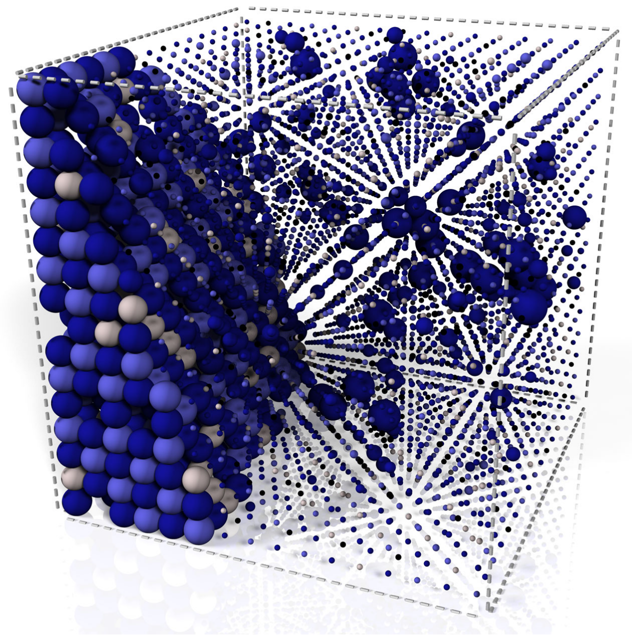


```
#!/bin/bash -x
#SBATCH -J STO_H128
#SBATCH --nodes=128
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=12
#SBATCH --time=07:00:00
#SBATCH --account=jara0172
##SBATCH --mem=28750
#SBATCH --mem-per-cpu=3750M
#SBATCH --exclusive

module load LIBRARIES hdf5

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/install/elpa-2018.05.001_intel19.0_intelmpi2018_skylake/lib
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export ELPA_DEFAULT_omp_threads=${SLURM_CPUS_PER_TASK}

srun $HOME/fleur_code/fleur_git/build/build_2019.08.16_devel_84c12a_CPU/build_SKL_intel19.0_intelmpi2018_ELPA201805
001_OldInt_hdf5/build/fleur_MPI
touch ready
```

SUMMARY

- HPC cluster
  - many nodes, each with many cores
- Parallel programming
  - two paradigms: shared vs. distributed memory
- FLEUR
  - part of the MaX (material design on exascale) project
  - three levels of parallelization:
    1) MPI over k-point
    2) MPI over eigenvectors
    3) OpenMP
  - can be run on HPC clusters
  - able to simulate large unit cells (thousands of atoms)



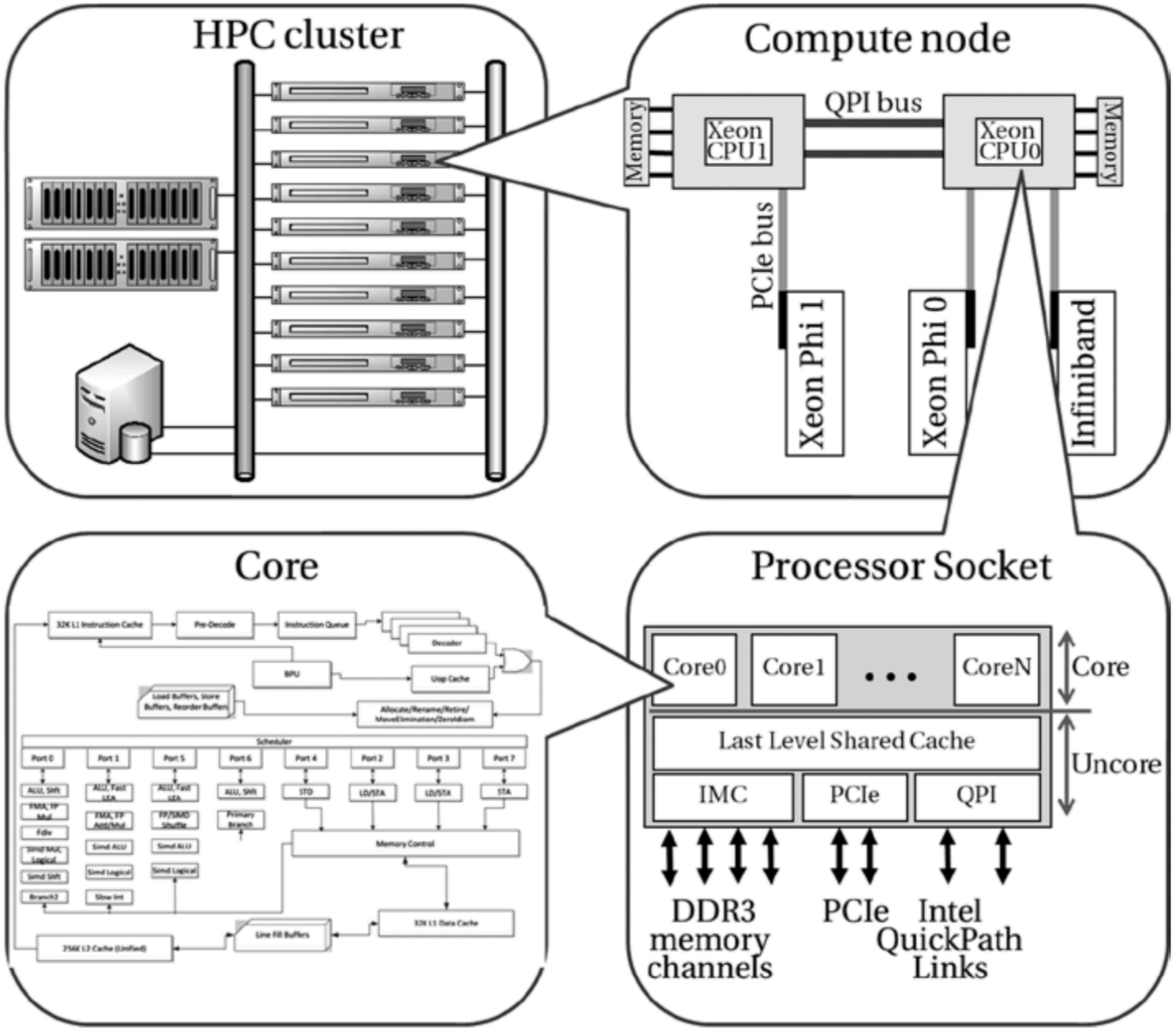


ARCHITECTURE

of a Modern HPC Cluster

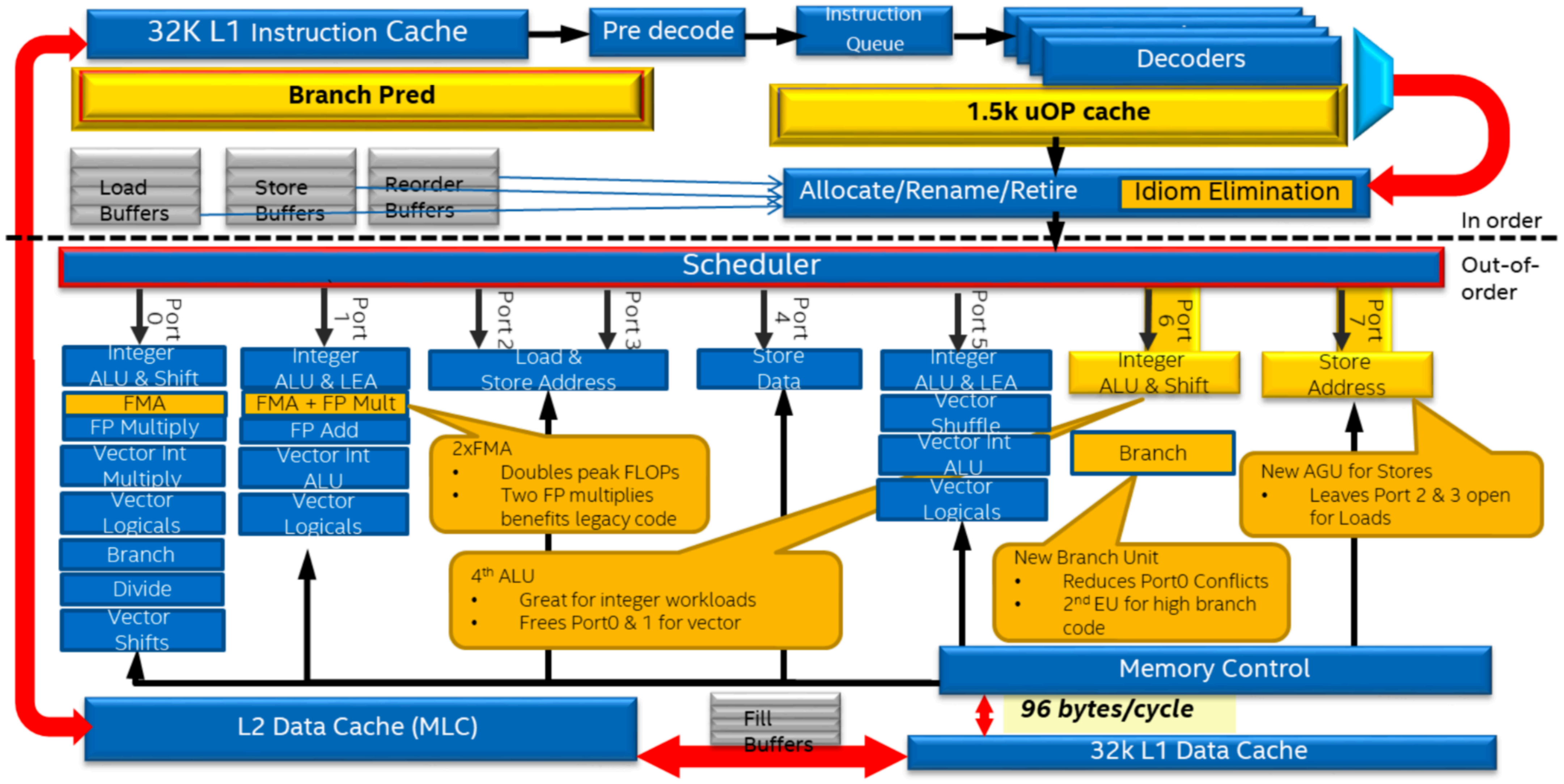
- cluster: many nodes
- node: several CPUs
- CPU: many cores
- core: few FPUs
- FPU: vectors

Memory: registers, caches
shared, distributed.



Supalov, Semin, Klemm, Dahnken
"Optimizing HPC applications with Intel Cluster Tools"

Haswell/Broadwell Microarchitecture


32K L1 Instruction Cache
Pre decode
Instruction Queue
Decoders
Branch Pred
1.5k uOP cache
Load Buffers
Store Buffers
Reorder Buffers
Allocate/Rename/Retire
Idiom Elimination
In order
Scheduler
Out-of-order
Port 0
Port 1
Port 2
Port 3
Port 4
Port 5
Port 6
Port 7
Integer ALU & Shift
FMA
FP Multiply
Vector Int Multiply
Vector Logicals
Branch
Divide
Vector Shifts
Integer ALU & LEA
FMA + FP Mult
FP Add
Vector Int ALU
Vector Logicals
Load & Store Address
Store Data
Integer ALU & LEA
Vector Shuffle
Vector Int ALU
Vector Logicals
Integer ALU & Shift
Branch
Store Address
2xFMA
• Doubles peak FLOPs
• Two FP multiplies benefits legacy code
4th ALU
• Great for integer workloads
• Frees Port0 & 1 for vector
New Branch Unit
• Reduces Port0 Conflicts
• 2nd EU for high branch code
New AGU for Stores
• Leaves Port 2 & 3 open for Loads
Memory Control
96 bytes/cycle
L2 Data Cache (MLC)
Fill Buffers
32k L1 Data Cache