UNIVERSITY OF Southampton

Oak Ridge User Conference

Technical Challenges for Ab-inito Simulations of Low and High-Pressure Turbines

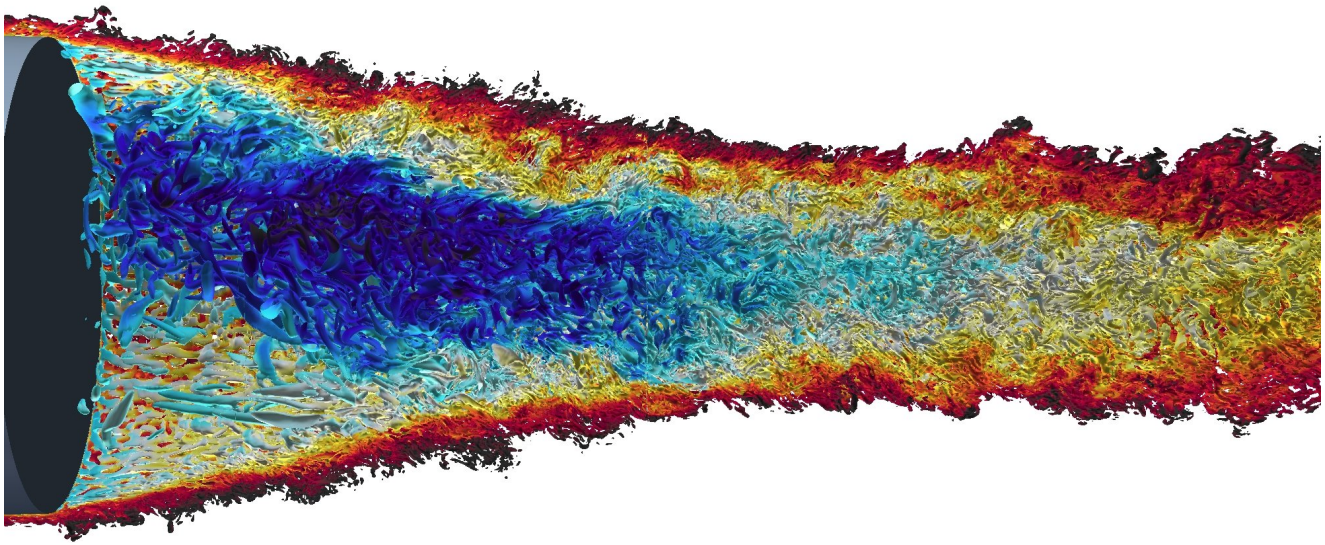
R. Pichler, R. D. Sandberg, V. Michelassi and G. Laskowski

Acknowledgements

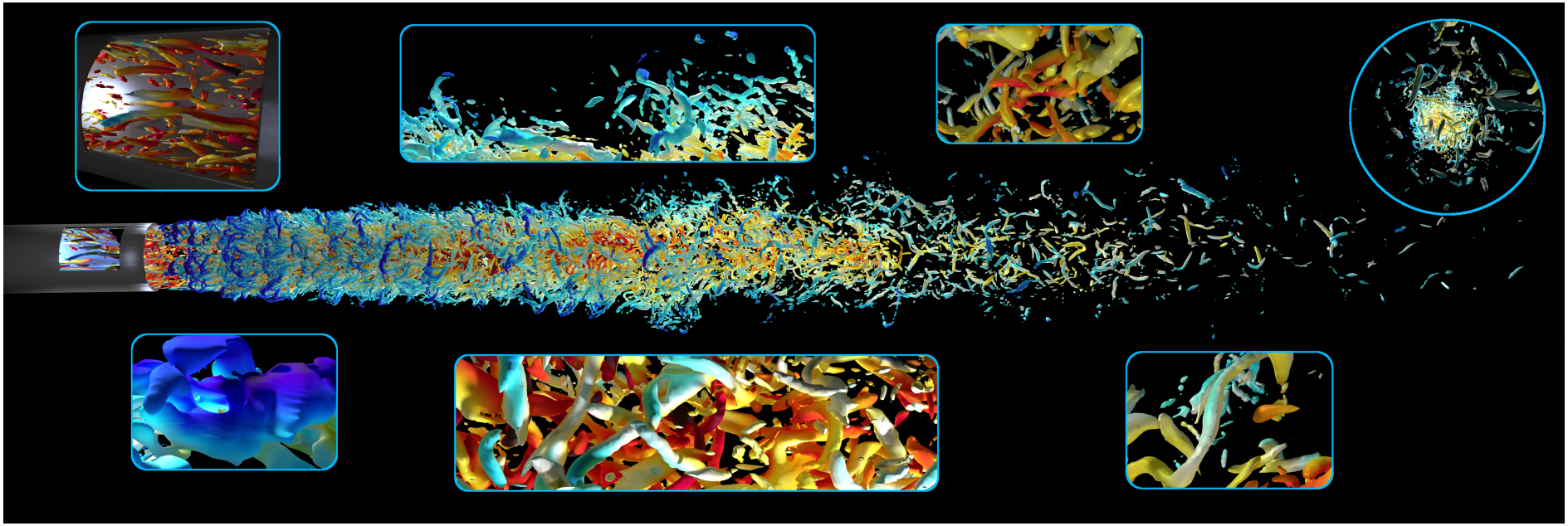
- ORNL Directors Grant - access to a GPU system
- John Levesque and Tom Edwards from CRAY COE - initial porting
- Jeff Larkin - Nvidia

Outline

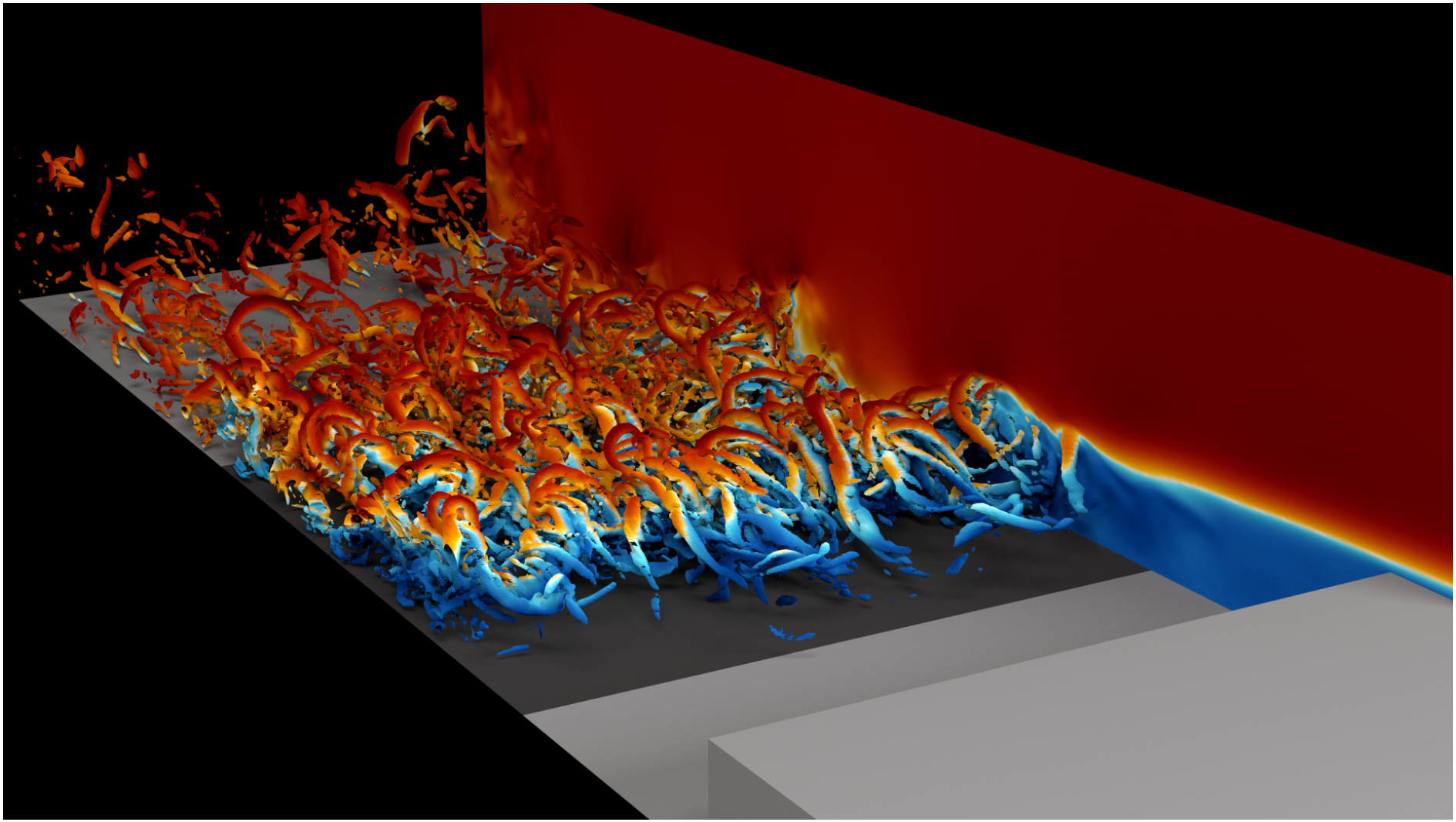
Research areas



Research areas



Research areas



HiPSTAR

High Performance Solver for Turbulence and Aeroacoustic Research

HiPSTAR

High Performance Solver for Turbulence and Aeroacoustic Research

Solves unsteady 3d compressible Flow equations - continuum

- Conservation of mass
- Conservation of momentum (3x)
- Conservation of energy

HiPSTAR

High Performance Solver for Turbulence and Aeroacoustic Research

Solves unsteady 3d compressible Flow equations - continuum

- Conservation of mass
- Conservation of momentum (3x)
- Conservation of energy
- Euler Frame of Reference
- neglect gravity

$$\frac{\partial}{\partial t}\left(\rho u_i\right) + \frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right) \tag{1}$$

$$\frac{\partial}{\partial t}\left(\rho \vec{u}\right) + \rho\left(\vec{u}\cdot\nabla\right)\vec{u} = -\nabla p + \nabla\cdot\underline{t} \tag{2}$$

Numerical Approach

$$\frac{\partial}{\partial t}\left(\rho u_i\right) + \frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right) \tag{3}$$

rewrite as

$$\frac{\partial}{\partial t}\left(\rho u_i\right) = \underbrace{-\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)}_{\textbf{RHS}} \tag{4}$$

Numerical Approach

$$\frac{\partial}{\partial t}\left(\rho u_i\right) + \frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right) \tag{3}$$

rewrite as

$$\frac{\partial}{\partial t}\left(\rho u_i\right) = \underbrace{-\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)}_{\mathbf{RHS}} \tag{4}$$

integrate numerically w.r.t time using explicit Runge-Kutta method

$$\rho u_i|_{n+1} = \rho u_i|_n + \Delta t \sum_{k=1}^{5} b_k \mathbf{RHS}|_n \tag{5}$$

Numerical Approach

$$\frac{\partial}{\partial t}\left(\rho u_i\right) + \frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right) \tag{3}$$

rewrite as

$$\frac{\partial}{\partial t}\left(\rho u_i\right) = \underbrace{-\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)}_{\mathbf{RHS}} \tag{4}$$

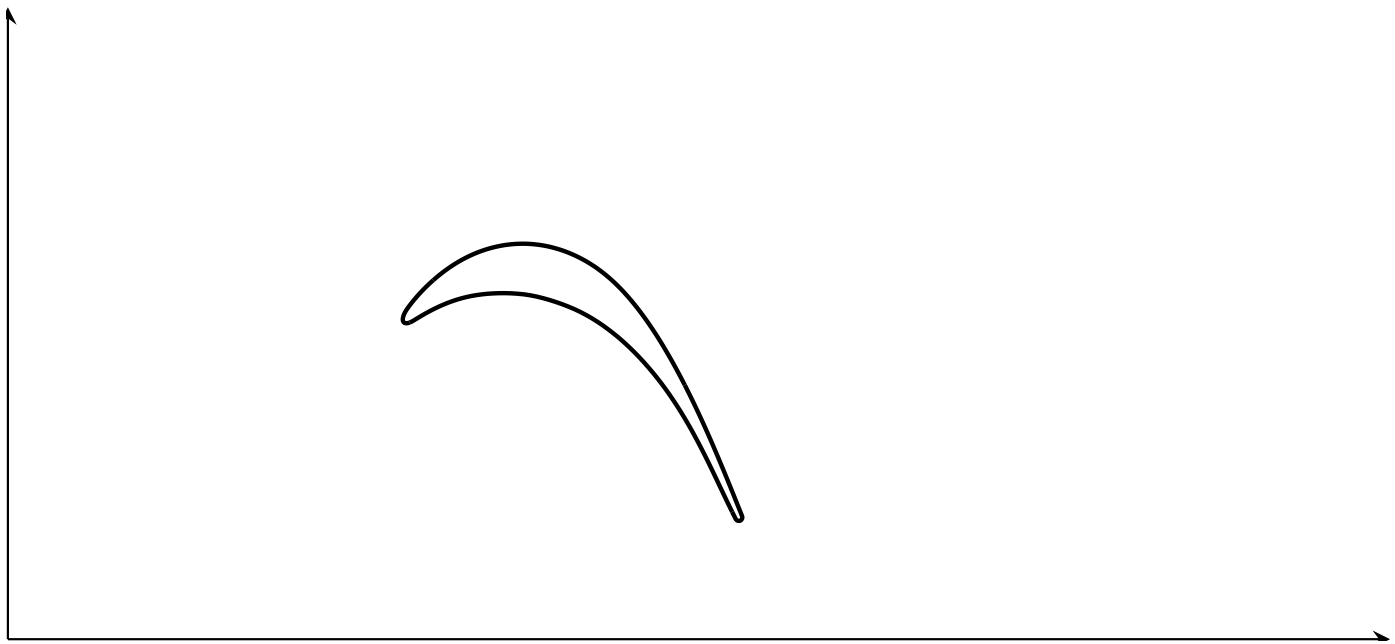
integrate numerically w.r.t time using explicit Runge-Kutta method

$$\rho u_i|_{n+1} = \rho u_i|_n + \Delta t \sum_{k=1}^{5} b_k \mathbf{RHS}|_n \tag{5}$$

spatial derivative discretized by 4^{th} order finite differences and Fourier method

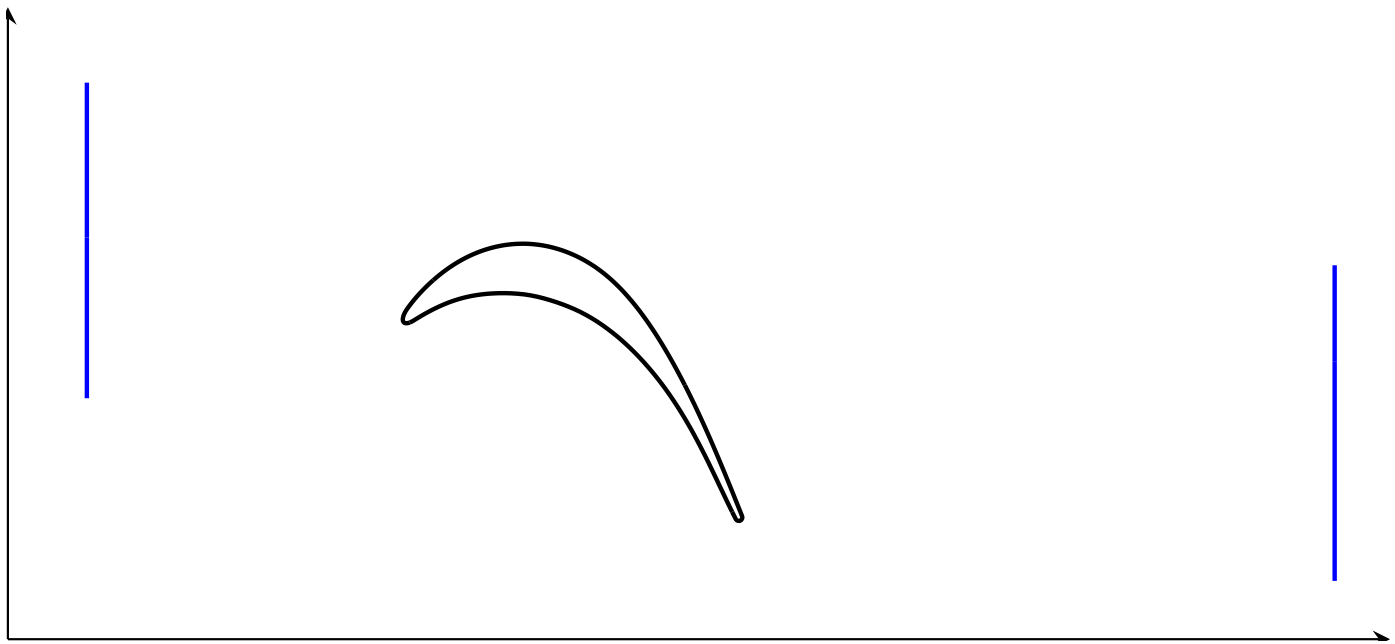
Parallelization

- discretize domain



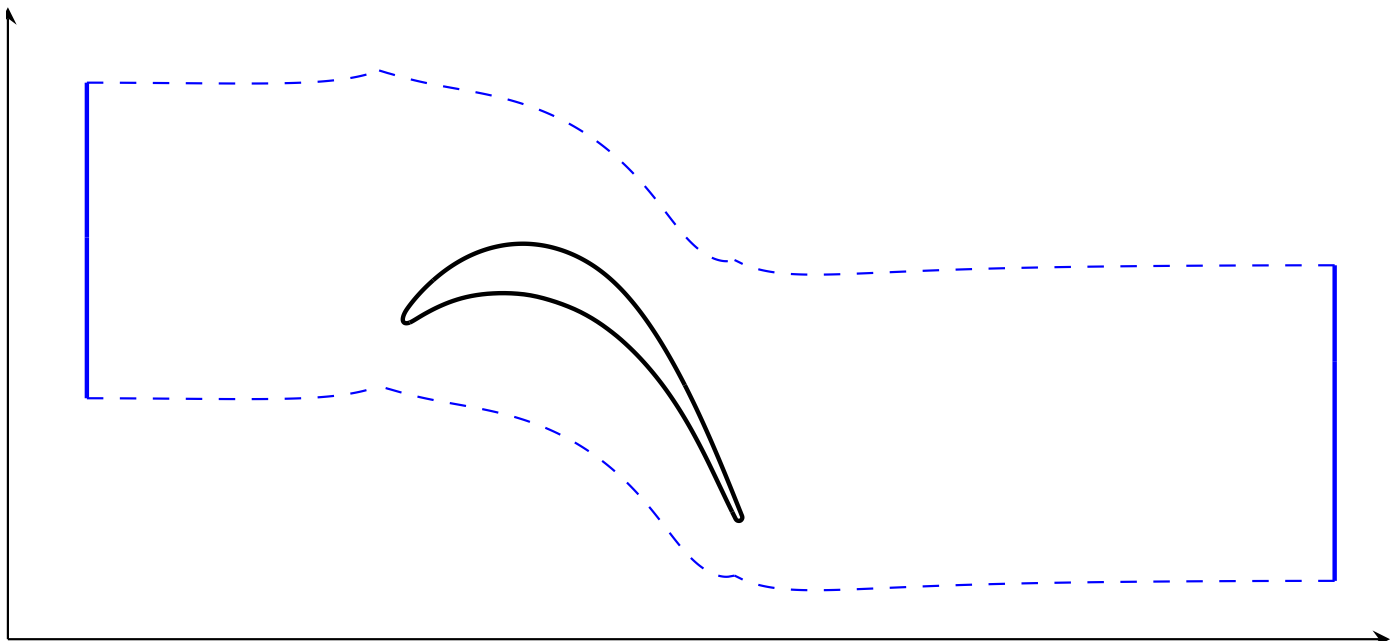
Parallelization

- discretize domain



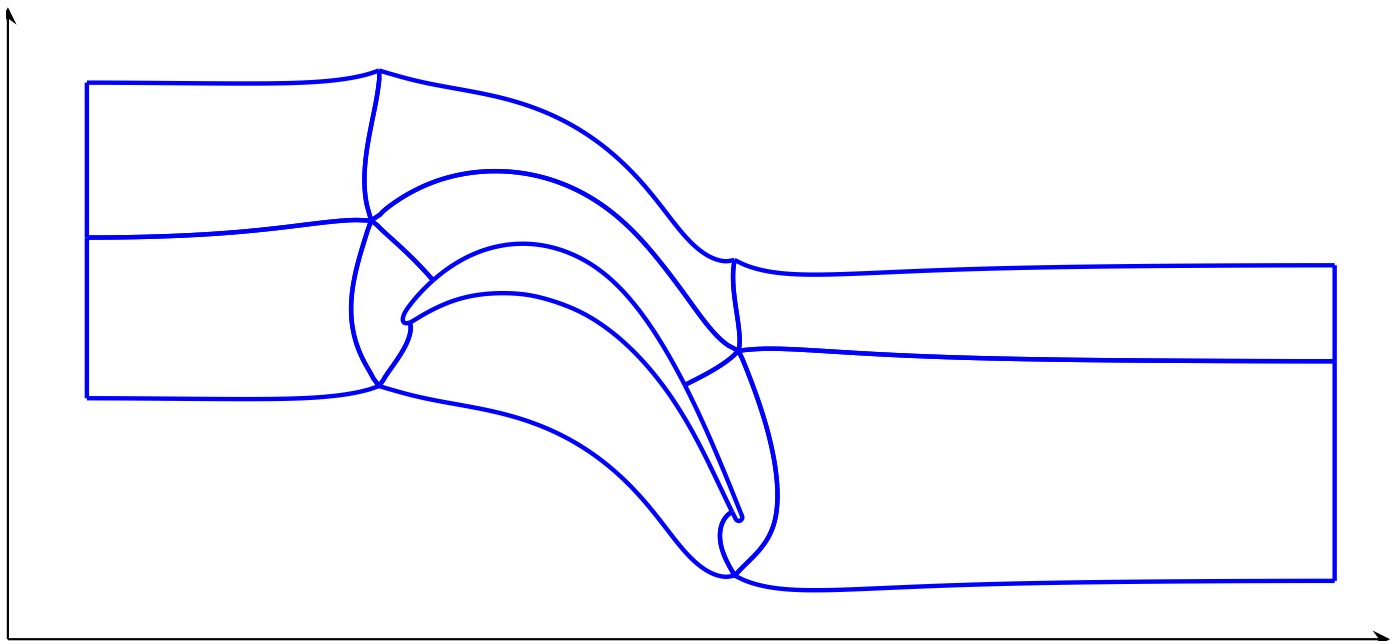
Parallelization

- discretize domain



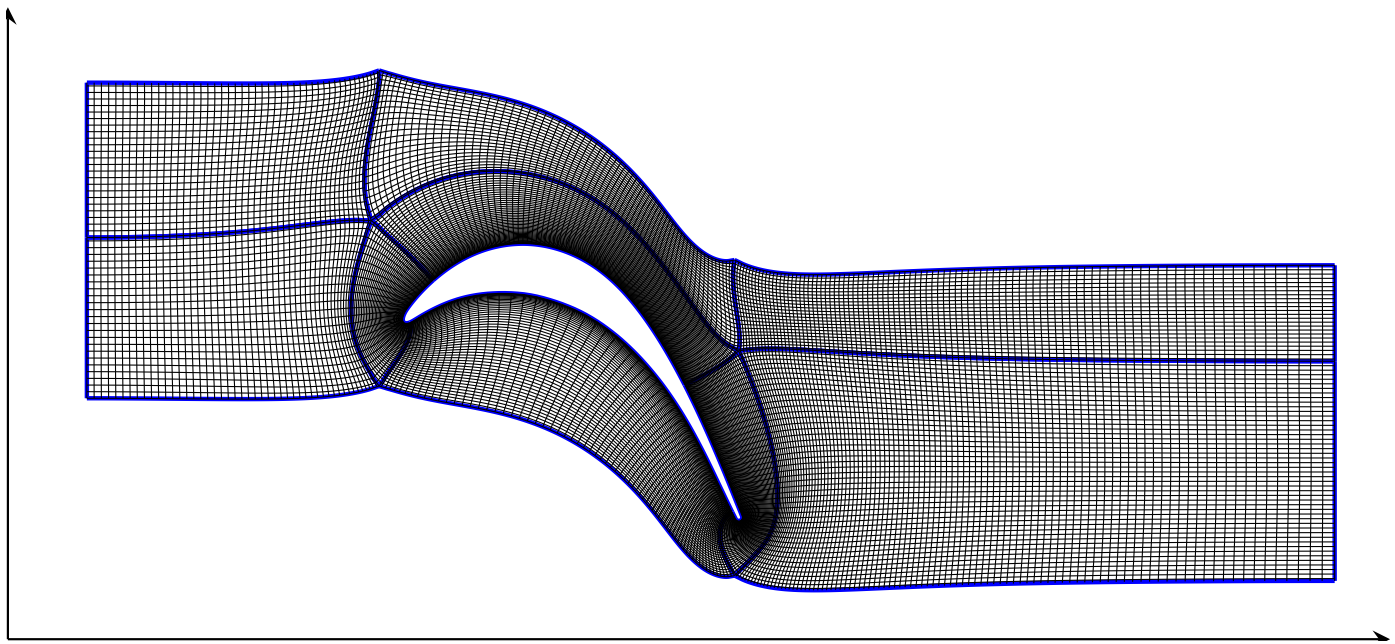
Parallelization

- discretize domain
- multi block grid for domain



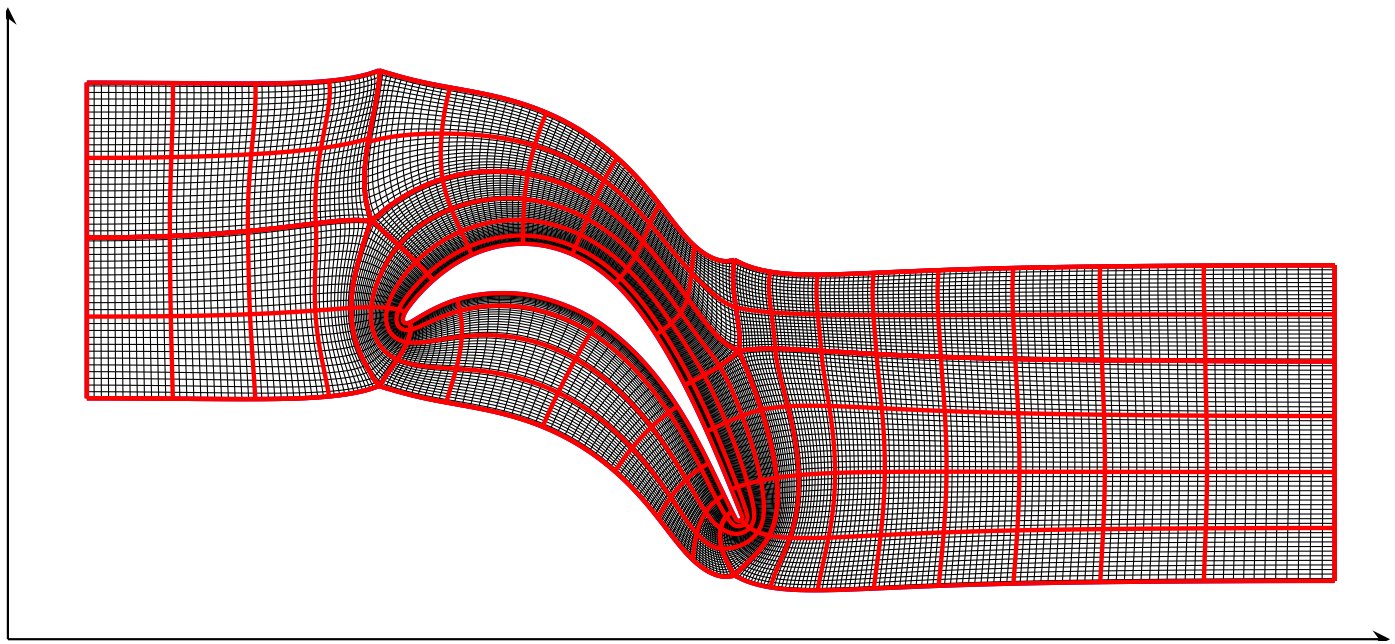
Parallelization

- discretize domain
- multi block grid for domain



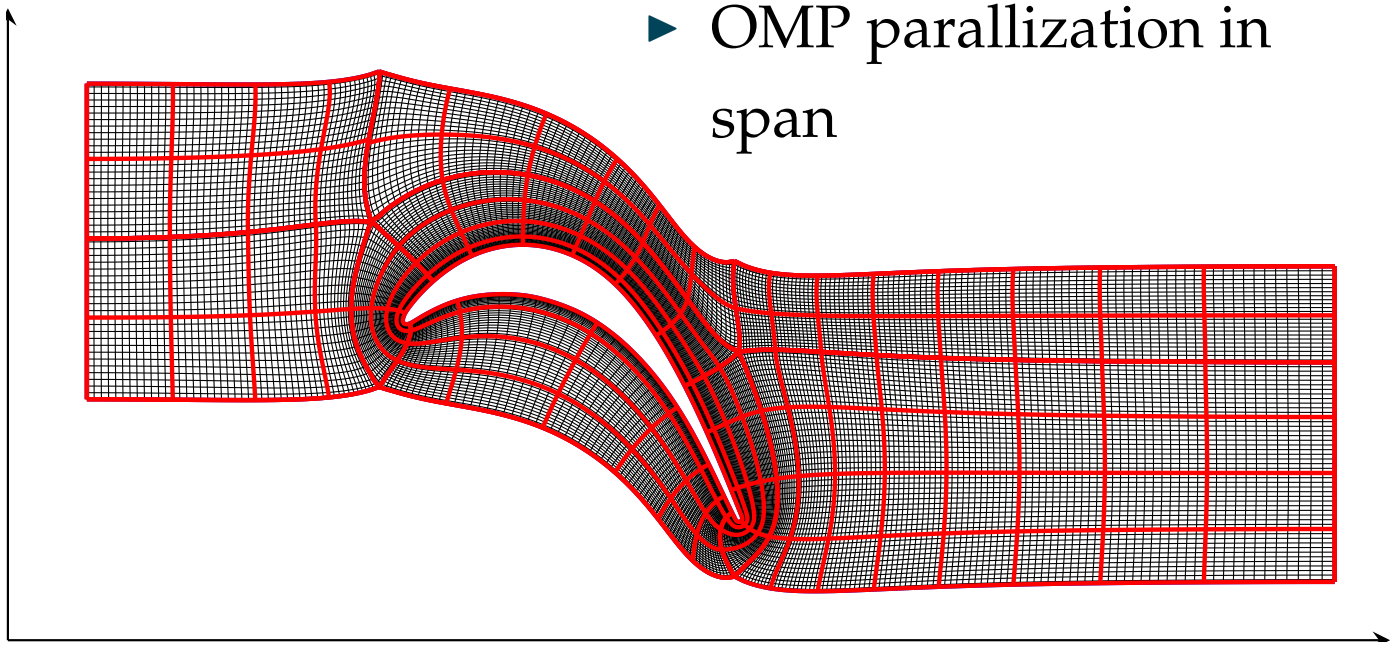
Parallelization

- discretize domain
- multi block grid for domain
- decompose grid using MPI

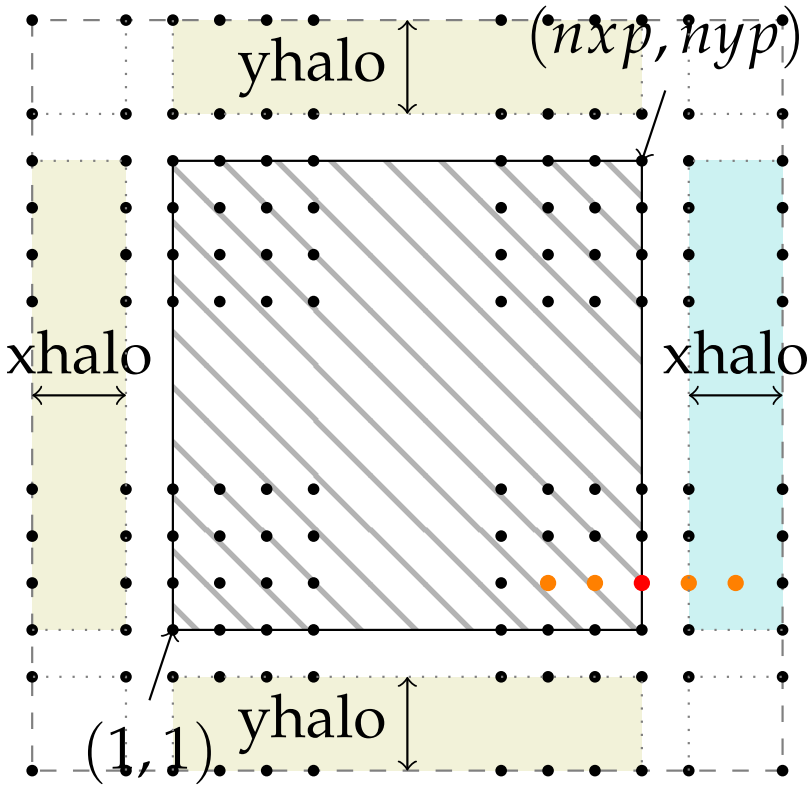


Parallelization

- discretize domain
- multi block grid for domain
- decompose grid using MPI
- OMP parallization in span

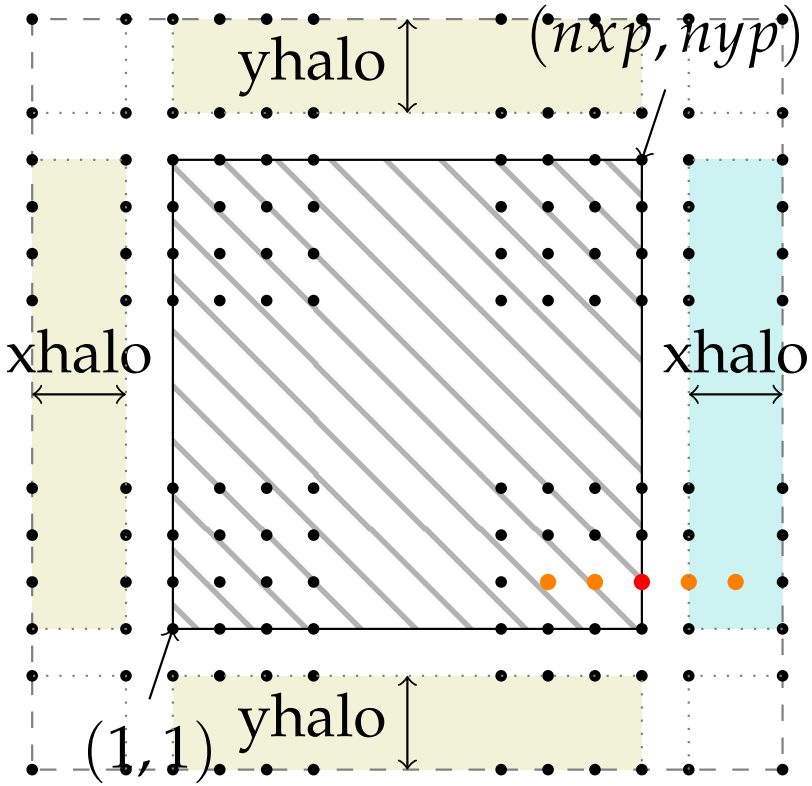


Domain decomposition



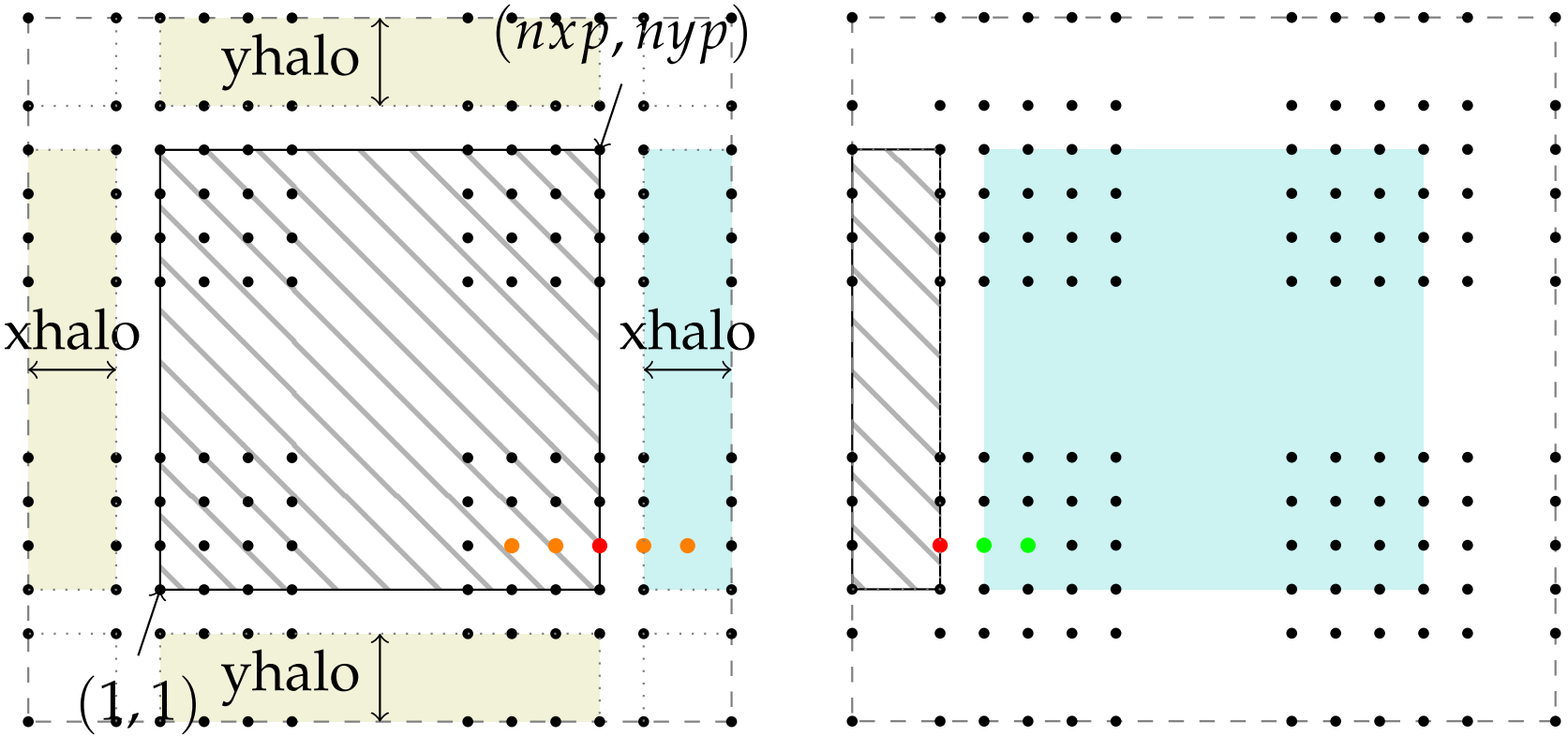
Domain decomposition

$$\frac{\partial}{\partial t}\left(\rho u_i\right) = \underbrace{-\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)}_{\textbf{RHS}}$$



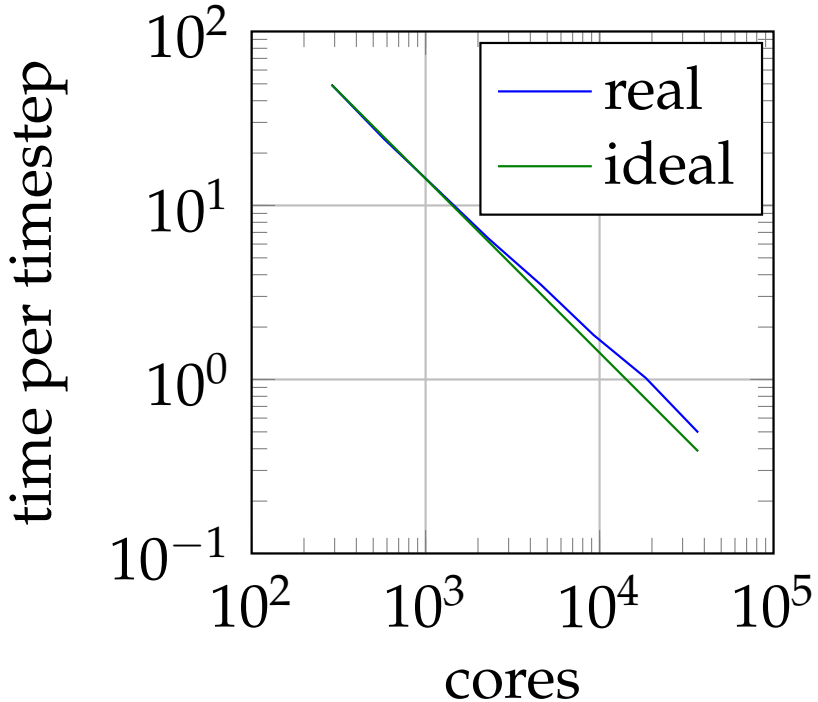
Domain decomposition

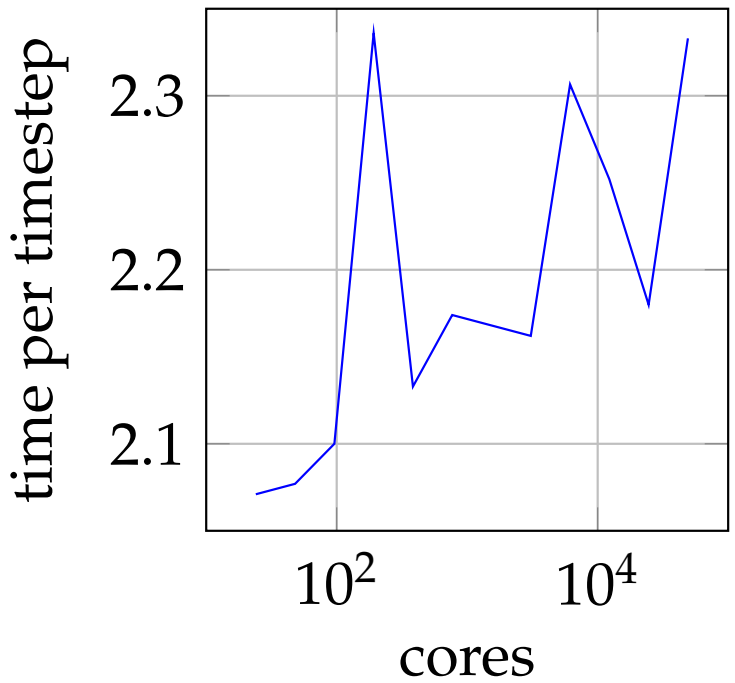
$$\frac{\partial}{\partial t}\left(\rho u_i\right) = \underbrace{-\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)}_{\textbf{RHS}}$$



CPU scaling - Plane jet

- Archer - UK national facility XC30
- strong scaling: speed up of 102 vs. 128
- weak scaling: 24 to 49152 cores





Code structure - CPU profile

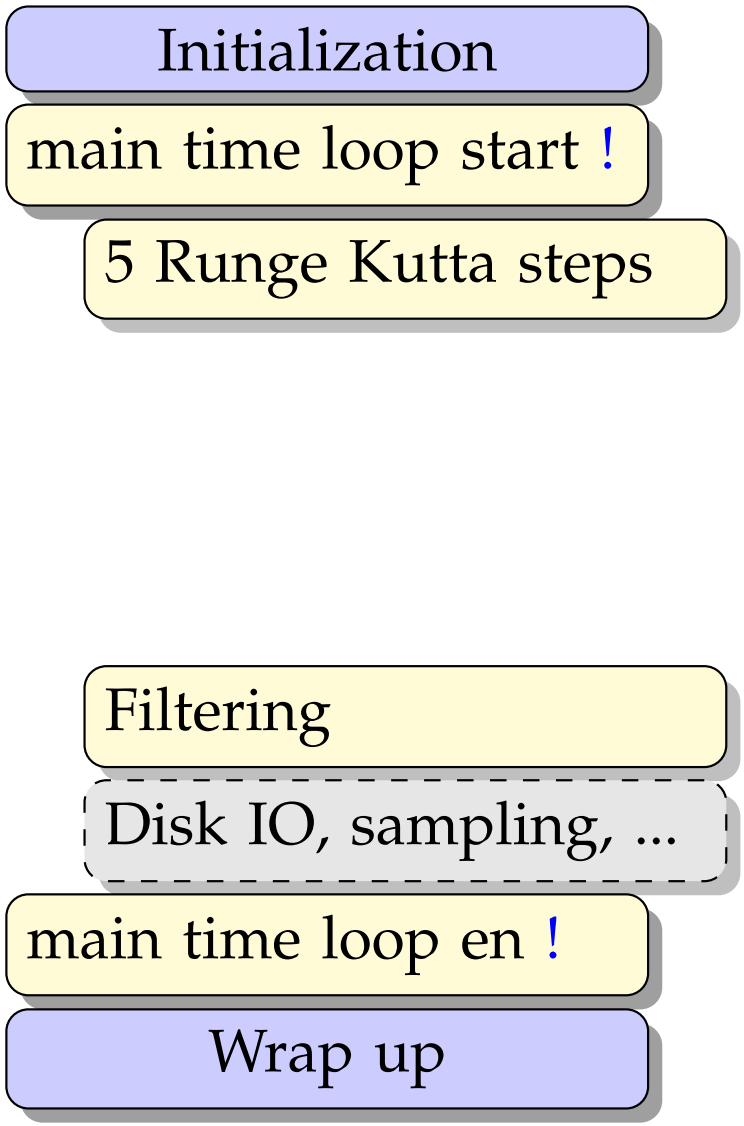
Initialization

main time loop start !

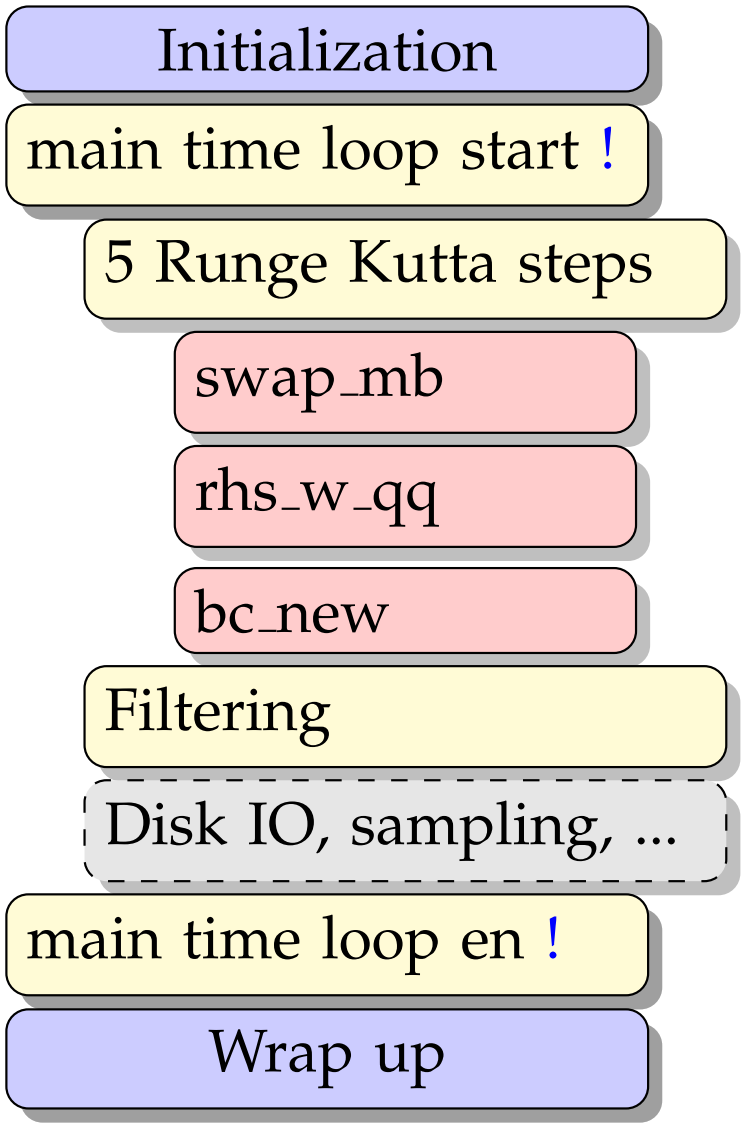
main time loop en !

Wrap up

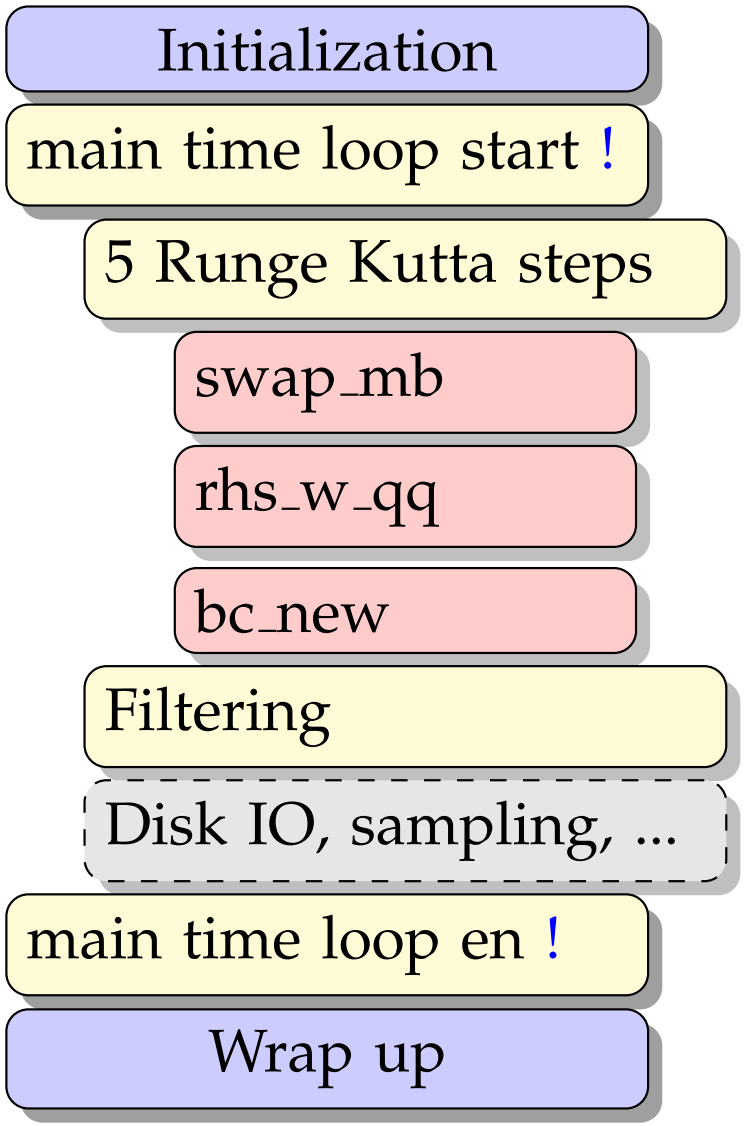
Code structure - CPU profile



Code structure - CPU profile



Code structure - CPU profile

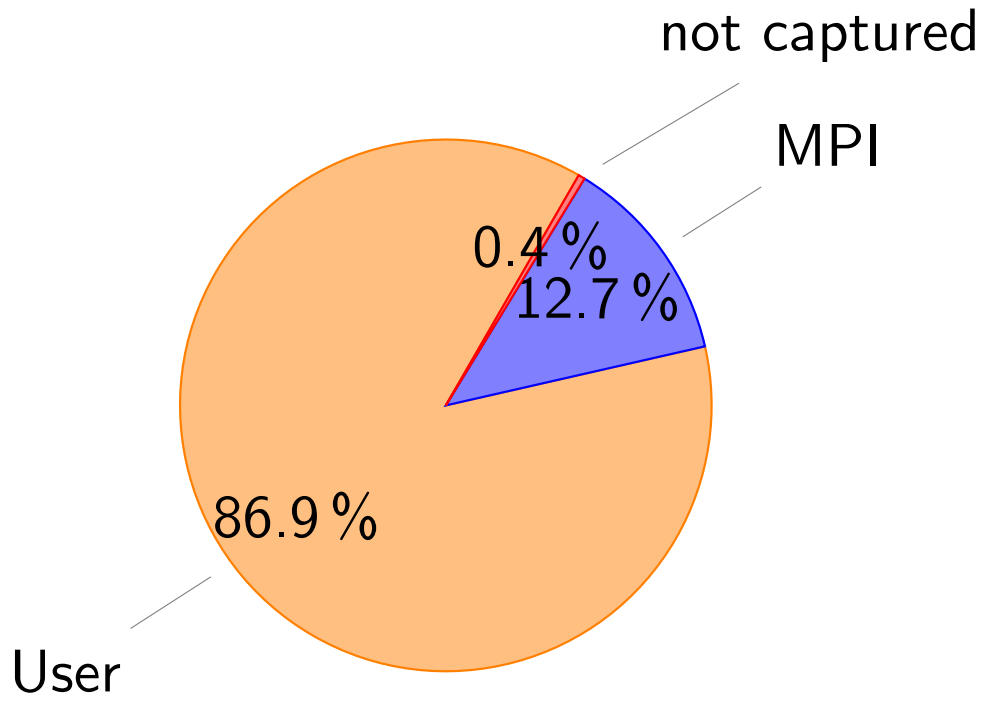


- Piz Daint - Xeon E5-2670 8C 2.600GHz
- CrayPAT - trace MPI and user functions in main loop

Code structure - CPU profile

- Initialization
- main time loop start !
  - 5 Runge Kutta steps
    - swap_mb
    - rhs_w_qq
    - bc_new
  - Filtering
  - Disk IO, sampling, ...
- main time loop en !
- Wrap up

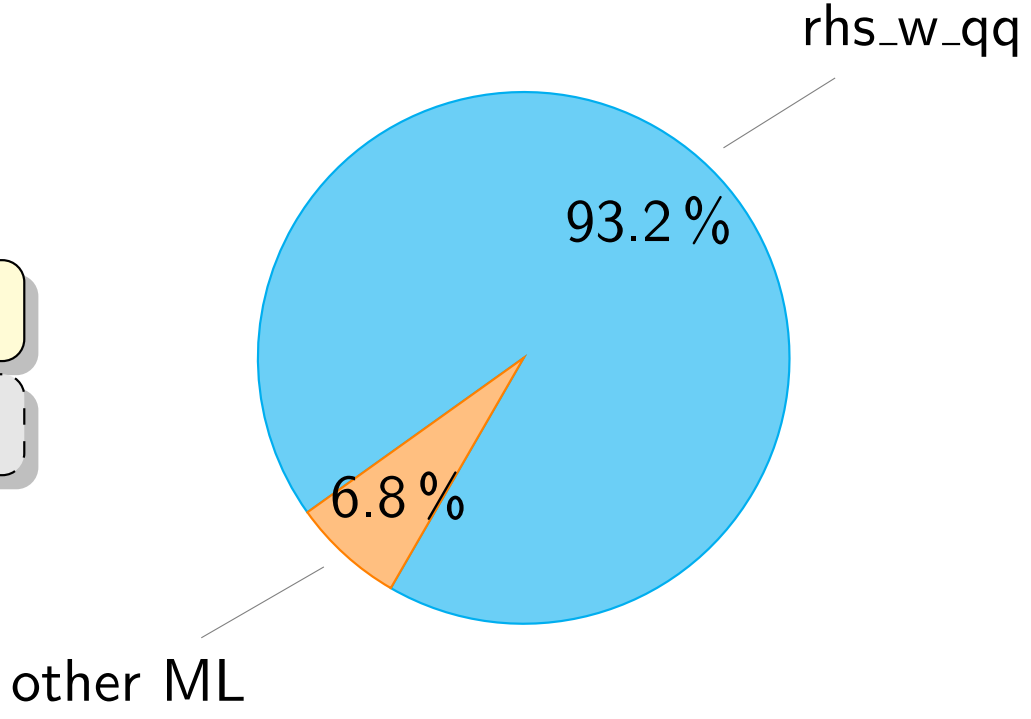
- Piz Daint - Xeon E5-2670 8C 2.600GHz
- CrayPAT - trace MPI and user functions in main loop



Code structure - CPU profile

- Initialization
- main time loop start !
  - 5 Runge Kutta steps
    - swap_mb
    - rhs_w_qq
    - bc_new
  - Filtering
  - Disk IO, sampling, ...
- main time loop en !
- Wrap up

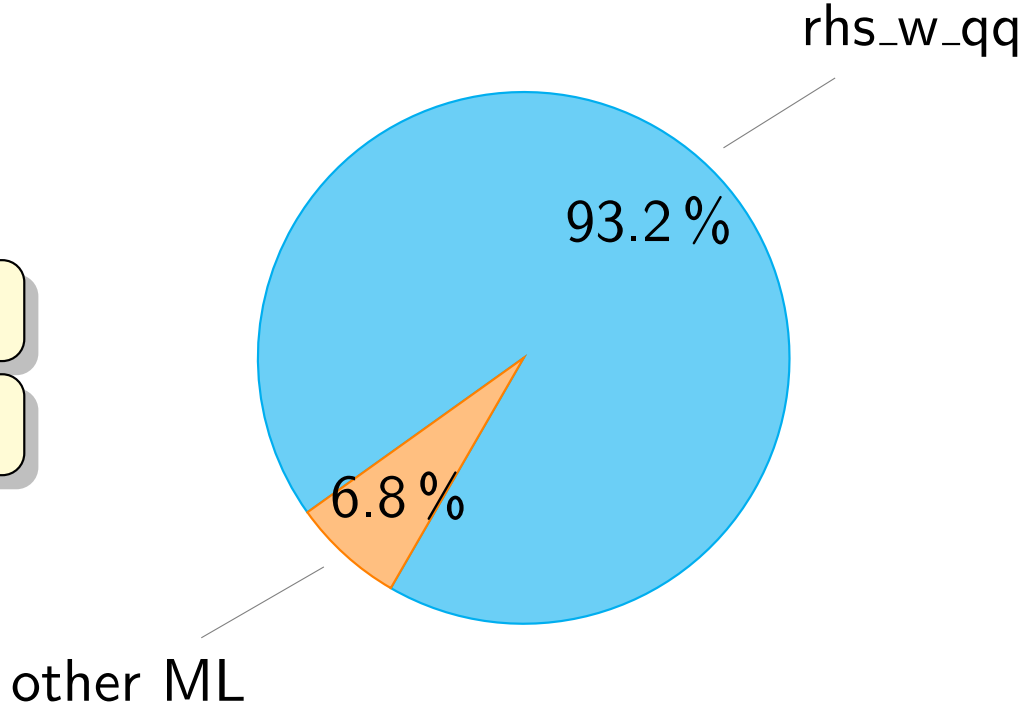
- Piz Daint - Xeon E5-2670 8C 2.600GHz
- CrayPAT - trace MPI and user functions in main loop



Code structure - CPU profile

- Initialization
- main time loop start !
  - 5 Runge Kutta steps
    - swap_mb
    - rhs_w_qq
    - bc_new
  - Filtering
  - Disk IO, sampling, ...
- main time loop en !
- Wrap up

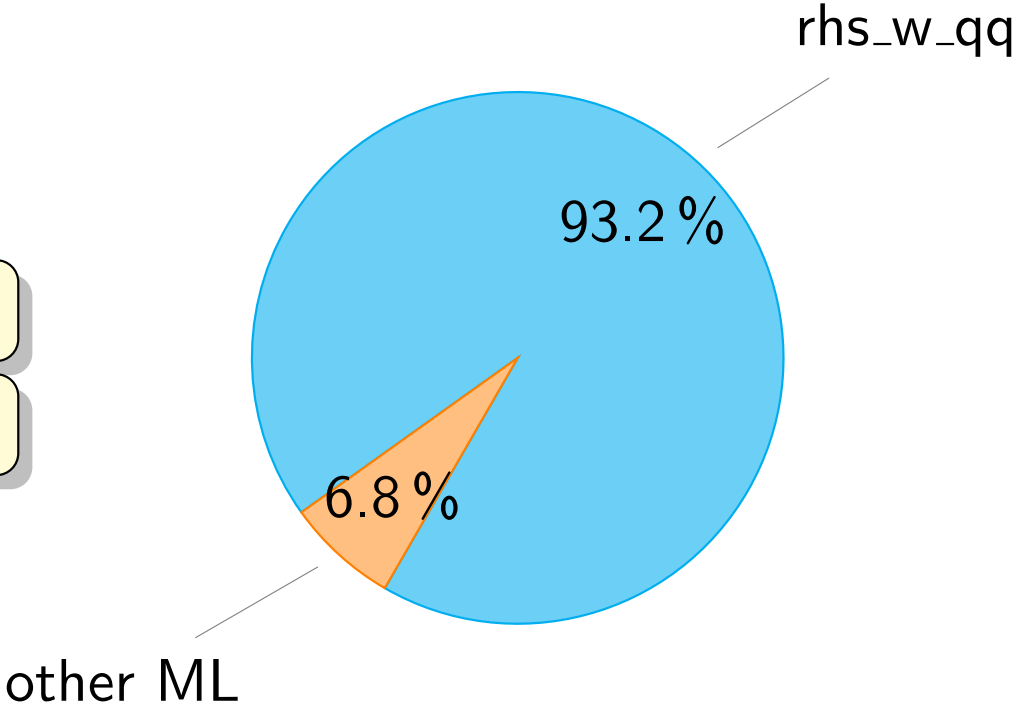
- Piz Daint - Xeon E5-2670 8C 2.600GHz
- CrayPAT - trace MPI and user functions in main loop



Code structure - CPU profile

- Initialization
- main time loop start !
  - 5 Runge Kutta steps
    - swap_mb
    - rhs_w_qq
    - bc_new
  - Filtering
  - Disk IO, sampling, ...
- main time loop en !
- Wrap up

- Piz Daint - Xeon E5-2670 8C 2.600GHz
- CrayPAT - trace MPI and user functions in main loop



1. step: port rhs_w_qq ⇒ data transfer

Step 1 - Computationally heavy Part

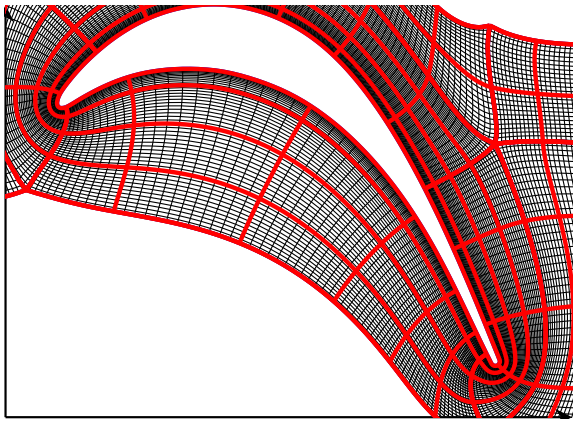
- “Only” 30% of the code have to be ported
- done by John Levesque and Tom Edwards (Cray)

Step 1 - Computationally heavy Part

- "Only" 30% of the code have to be ported
- done by John Levesque and Tom Edwards (Cray)
- add ACC statements to loops
- define data regions $\Rightarrow$ CPU-GPU data transfer

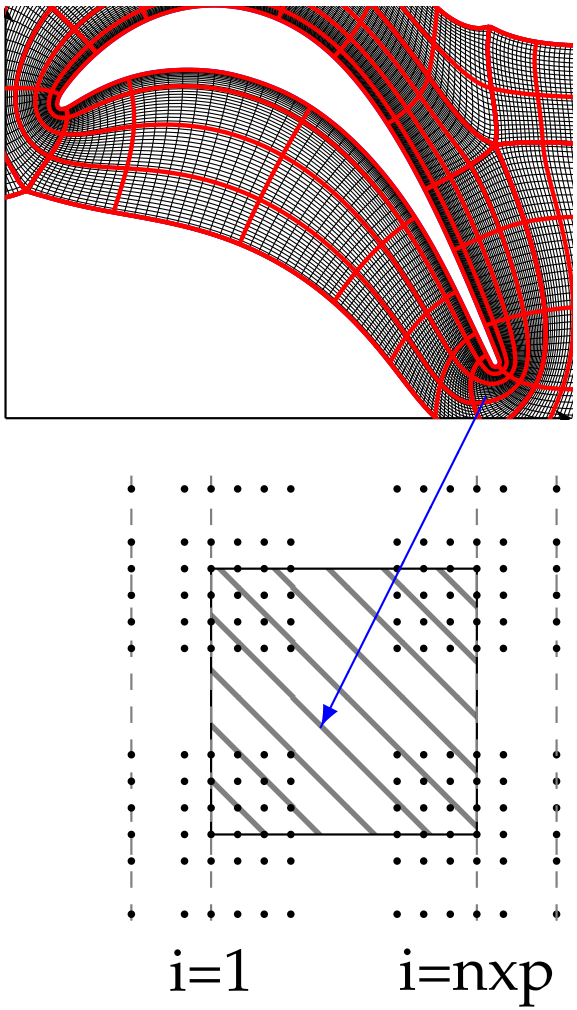
Step 1 - Computationally heavy Part

$$\mathbf{RHS} = -\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)$$



Step 1 - Computationally heavy Part

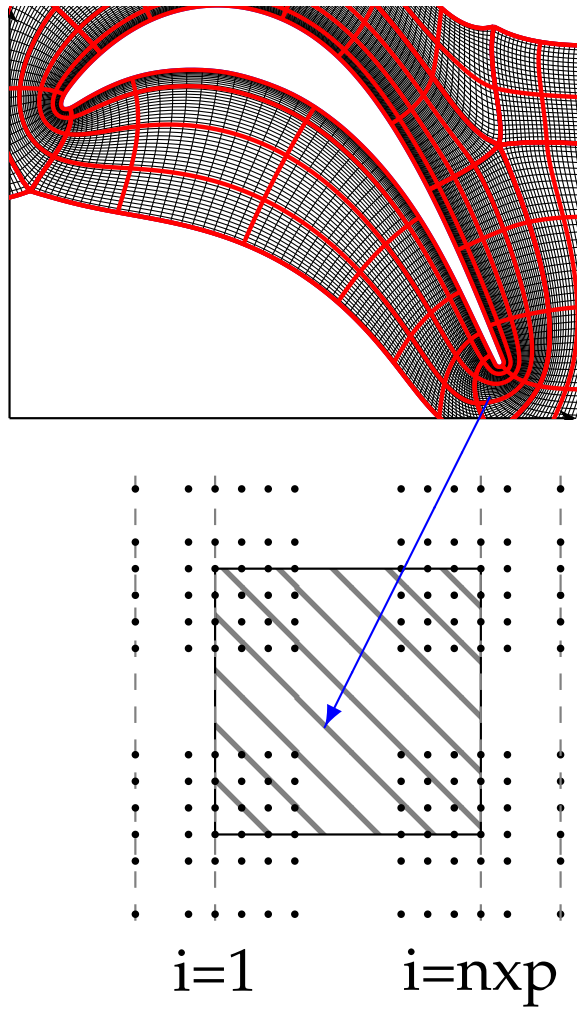
$$\mathbf{RHS} = -\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)$$



Step 1 - Computationally heavy Part

$$\mathbf{RHS} = -\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)$$

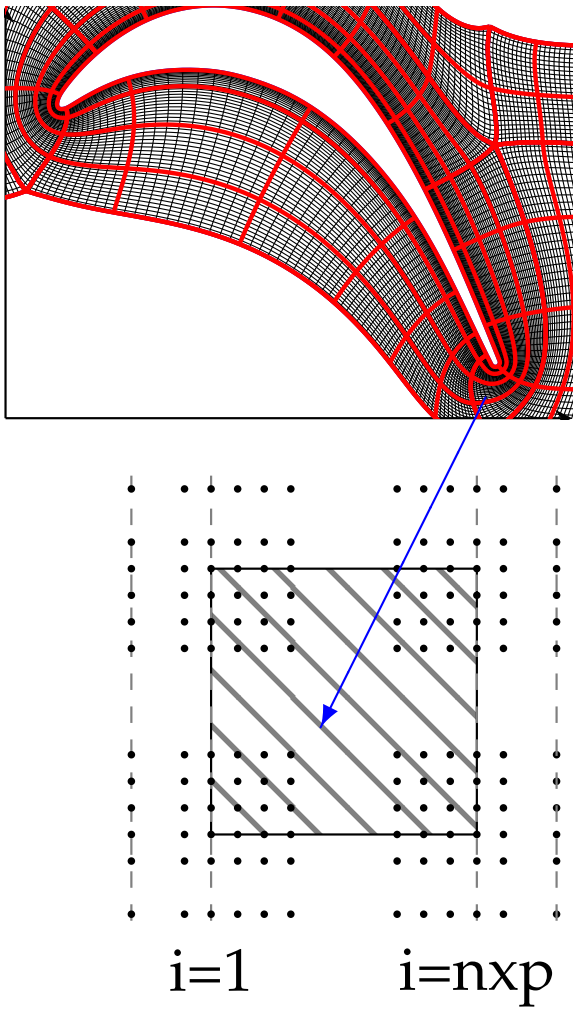
```
do k=nzp_start,nzp_end

  do j=jbegwp,jendwp

    do i=ibegwp,iendwp
      ddw4(i,j,1) = qph(i,j,k,2)*r_et(i,j)
    end do
  end do


  call ddxi(ddw4(1-xhalo,1-yhalo,1)
&           ,ddw1(1-xhalo,1-yhalo,1) ,1)


  do j=1,nyp

    do i=1,nxp
      q(i,j,k,2)= q(i,j,k,2)+ddw1(i,j,1)
    end do
  end do
end do
```



Step 1 - Computationally heavy Part

$$\mathbf{RHS} = -\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)$$

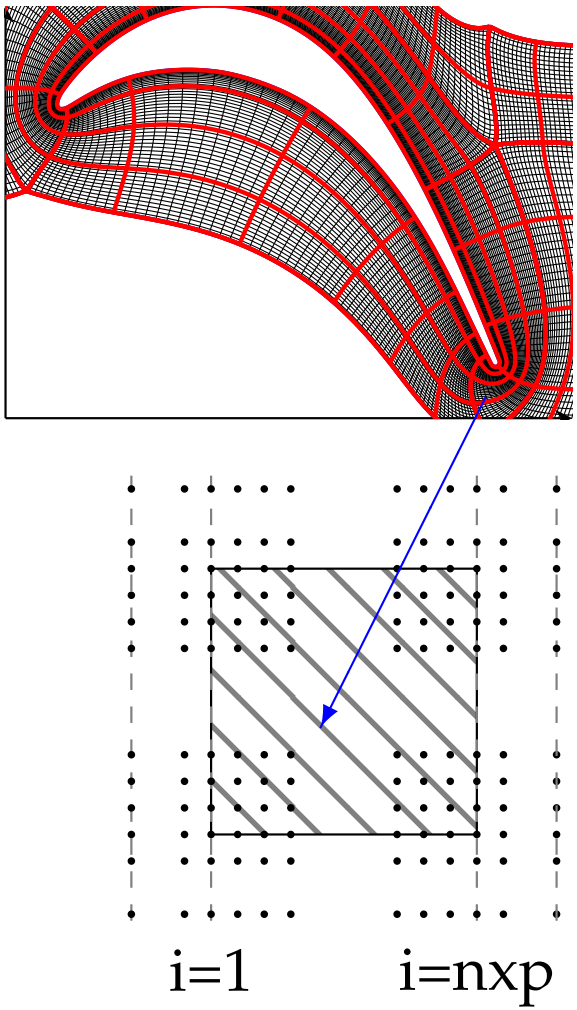
```
do k=nzp_start,nzp_end

  do j=jbegwp,jendwp

    do i=ibegwp,iendwp
      ddw4(i,j,k) = qph(i,j,k,2)*r_et(i,j)
    end do
  end do
end do

  call ddxi_all(ddw4(1-xhalo,1-yhalo,1)
&             ,ddw1(1-xhalo,1-yhalo,1) ,nzp)

do k=nzp_start,nzp_end

  do j=1,nyp

    do i=1,nxp
      q(i,j,k,2)= q(i,j,k,2)+ddw1(i,j,k)
    end do
  end do
end do
```



Step 1 - Computationally heavy Part

```
!$acc parallel loop gang
      do k=nzp_start,nzp_end
!$acc loop worker
        do j=jbegwp,jendwp
!$acc loop vector
          do i=ibegwp,iendwp
            ddw4(i,j,k) = qph(i,j,k,2)*r_et(i,j)
          end do
        end do
      end do
!$acc end parallel loop
        call ddxi_all(ddw4(1-xhalo,1-yhalo,1)
     &              ,ddw1(1-xhalo,1-yhalo,1)  ,nzp)
!$acc parallel loop gang
      do k=nzp_start,nzp_end
!$acc loop worker
        do j=1,nyp
!$acc loop vector
          do i=1,nxp
            q(i,j,k,2)= q(i,j,k,2)+ddw1(i,j,k)
          end do
        end do
      end do
!$acc end parallel loop
```

$$\mathbf{RHS} = -\frac{\partial}{\partial x_j}\left(\rho u_j u_i\right) - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(t_{ji}\right)$$



Step 1 - Computationally heavy Part

- compare runtime of
  - hybrid open MPI/OMP with 1 MPI rank per node (16 OMP threads)
  - ACC version using 1 MPI rank per node

Step 1 - Computationally heavy Part

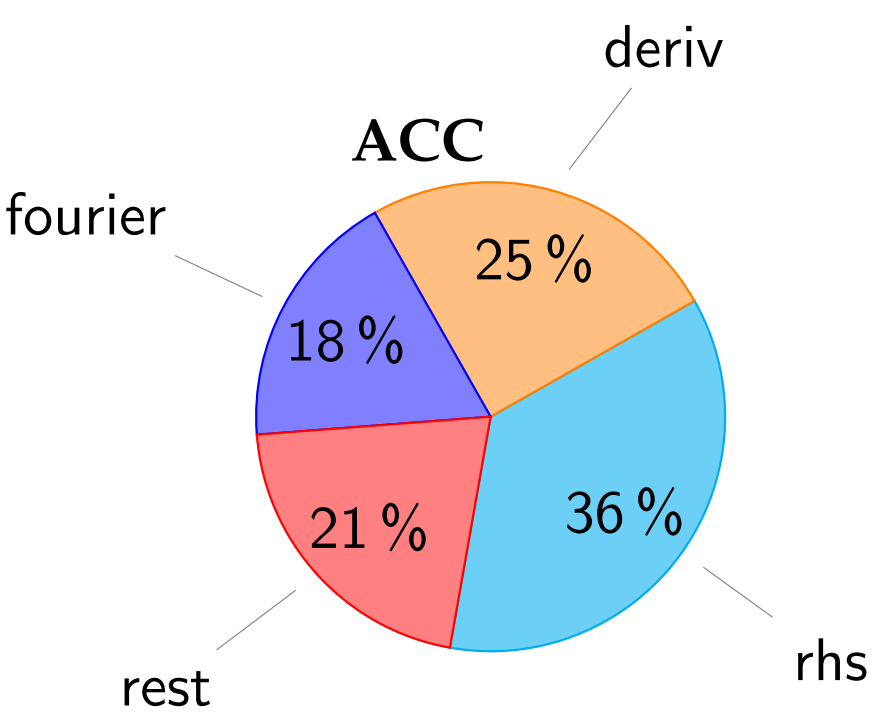
- compare runtime of
  - hybrid open MPI/OMP with 1 MPI rank per node (16 OMP threads)
  - ACC version using 1 MPI rank per node
- overall speedup of 18% (GPU vs. CPU)

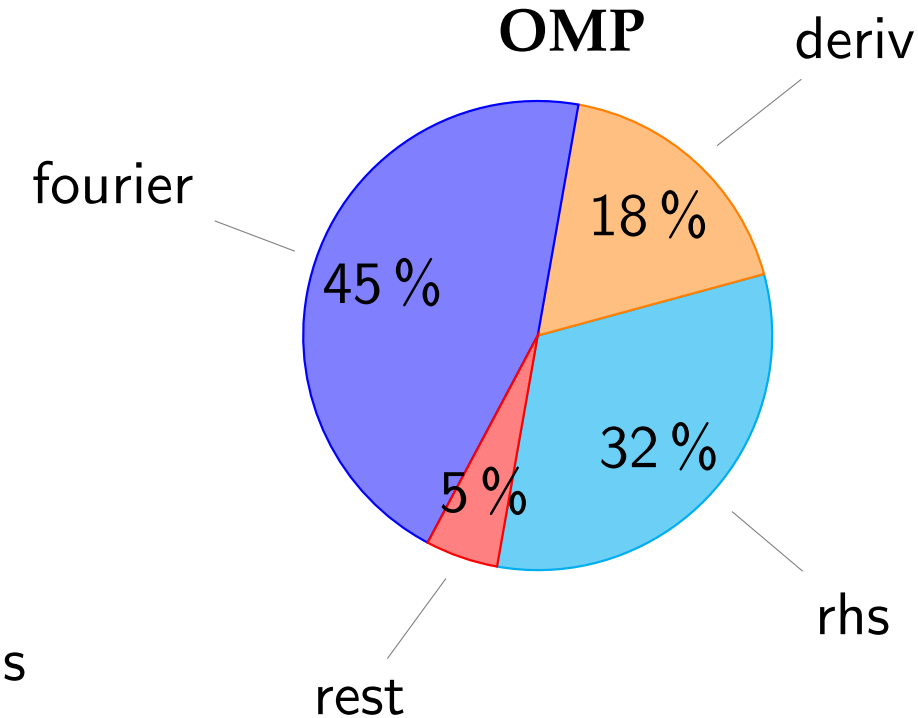
Step 1 - Computationally heavy Part

- compare runtime of
  - hybrid open MPI/OMP with 1 MPI rank per node (16 OMP threads)
  - ACC version using 1 MPI rank per node
- overall speedup of 18% (GPU vs. CPU)
- data transfer between GPU and CPU $\approx$ 30% of runtime

Step 1 - Computationally heavy Part

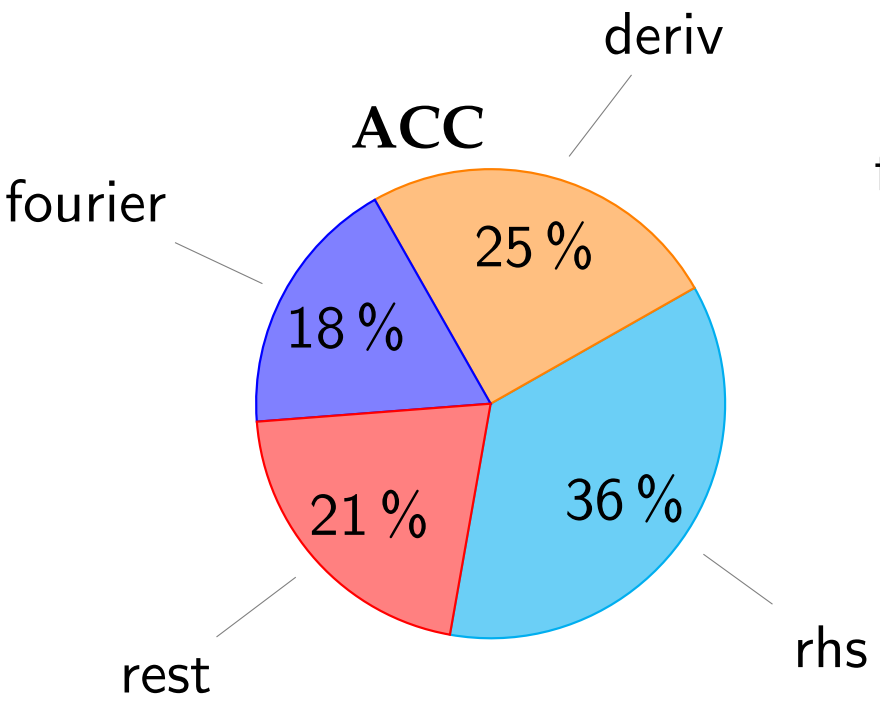
- *rest* $\Rightarrow$ CPU code runs slower (1 vs 16 OMP threads)

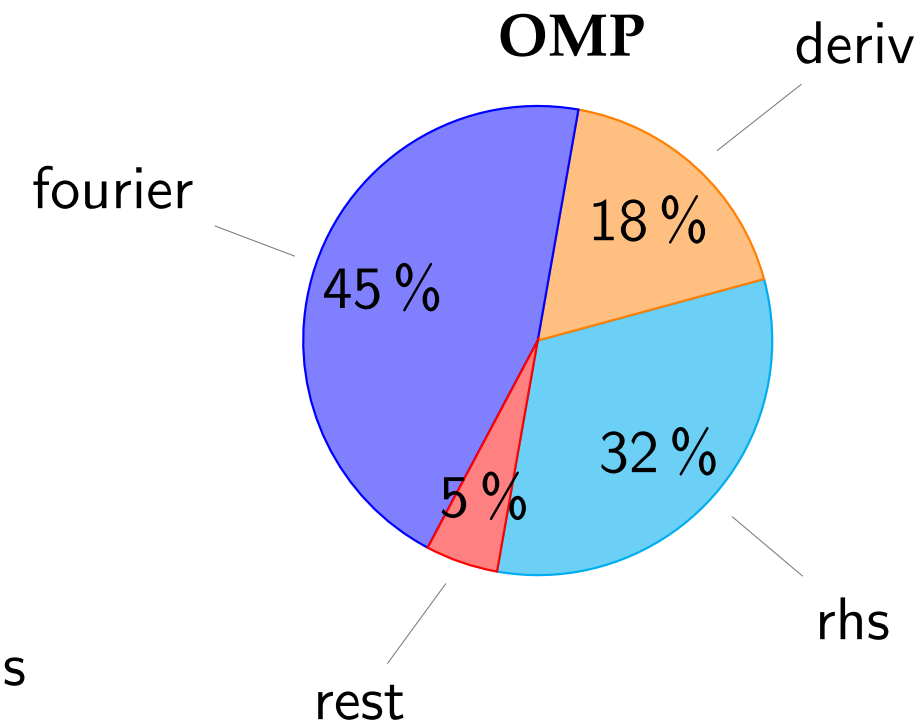




Step 1 - Computationally heavy Part

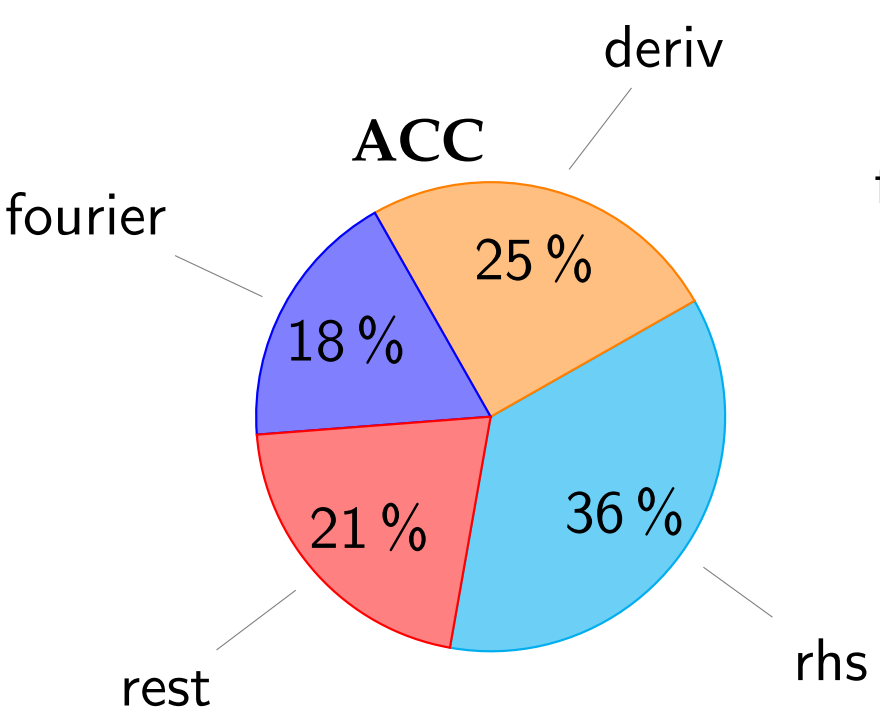
- *rest* $\Rightarrow$ CPU code runs slower (1 vs 16 OMP threads)
- FFTW (cufft) are significantly faster

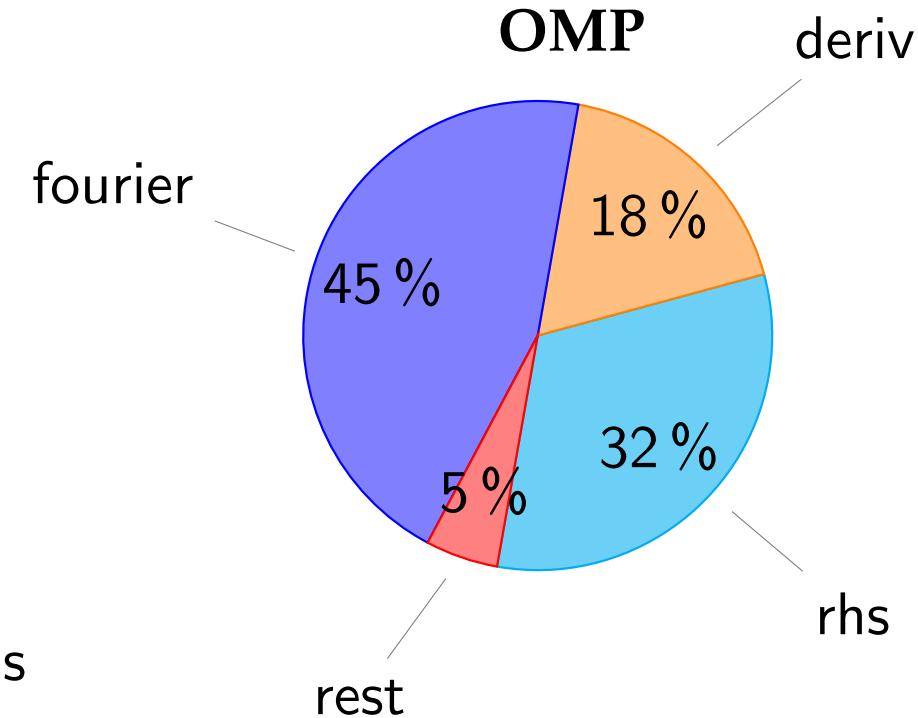




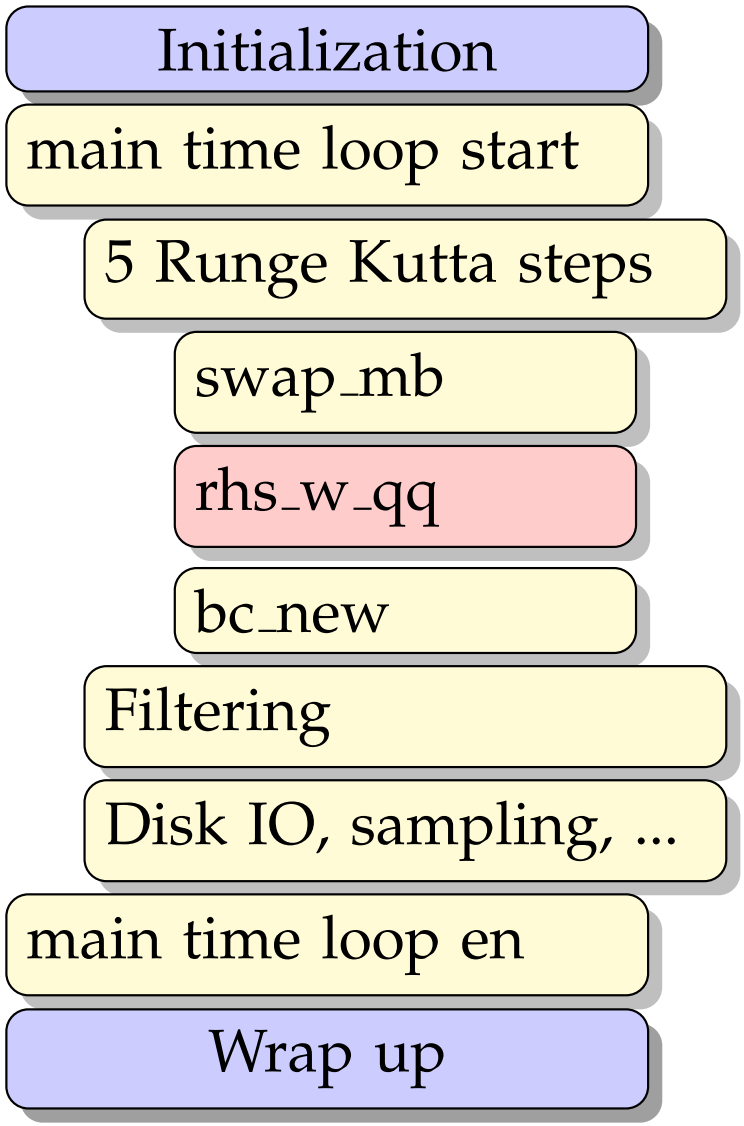
Step 1 - Computationally heavy Part

- *rest* $\Rightarrow$ CPU code runs slower (1 vs 16 OMP threads)
- FFTW (cufft) are significantly faster
- *deriv* and *rhs* similar relative fraction



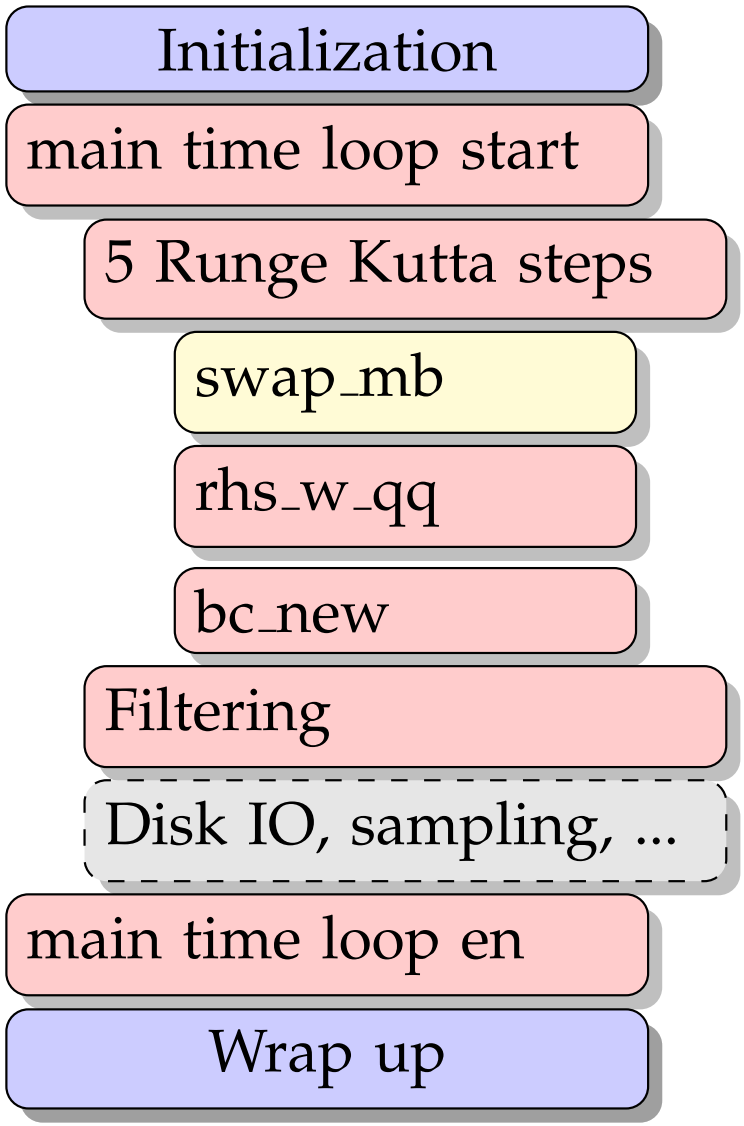


Step 2 - Limitting data transfer



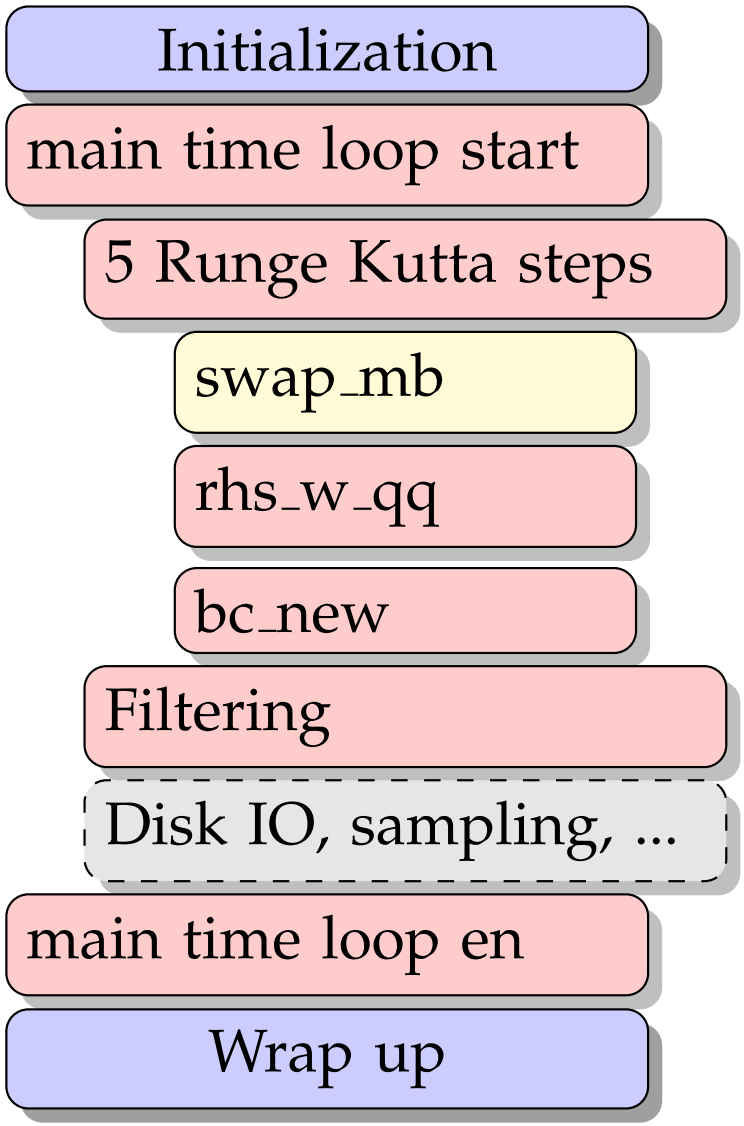
- 30% of walltime spent in CPU-GPU data transfer
- move everything to GPU except for

Step 2 - Limitting data transfer



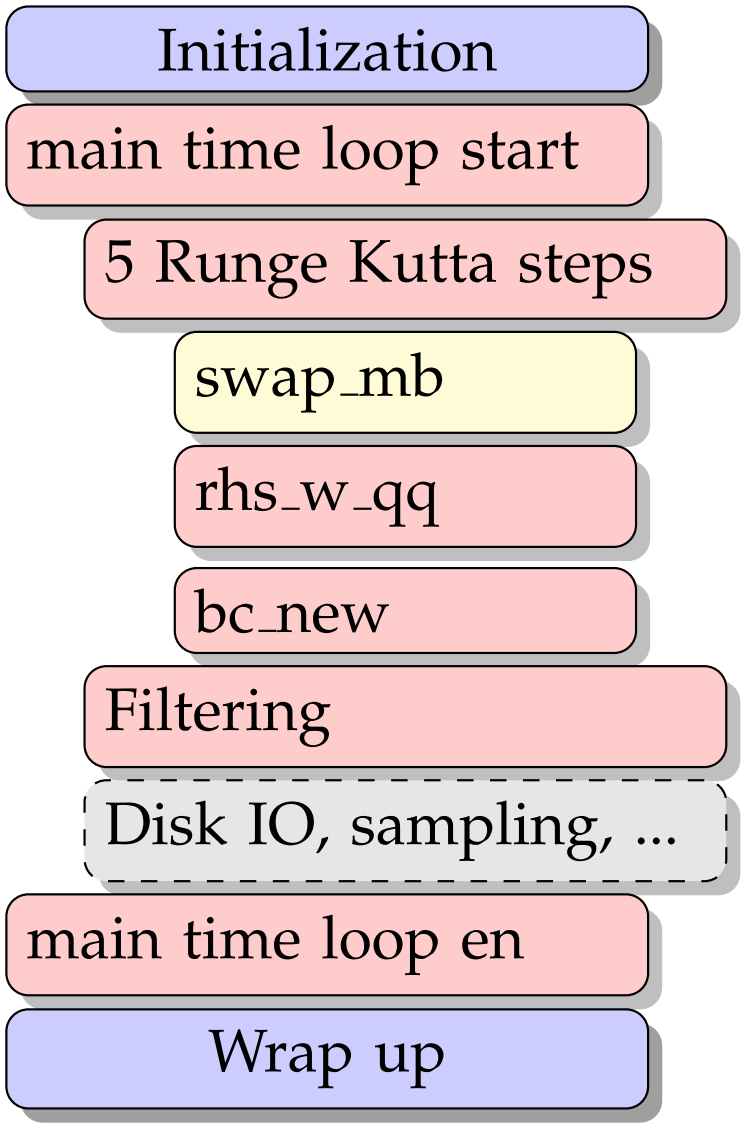
- 30% of walltime spent in CPU-GPU data transfer
- move everything to GPU except for
  - swapping - MPI communication

Step 2 - Limitting data transfer



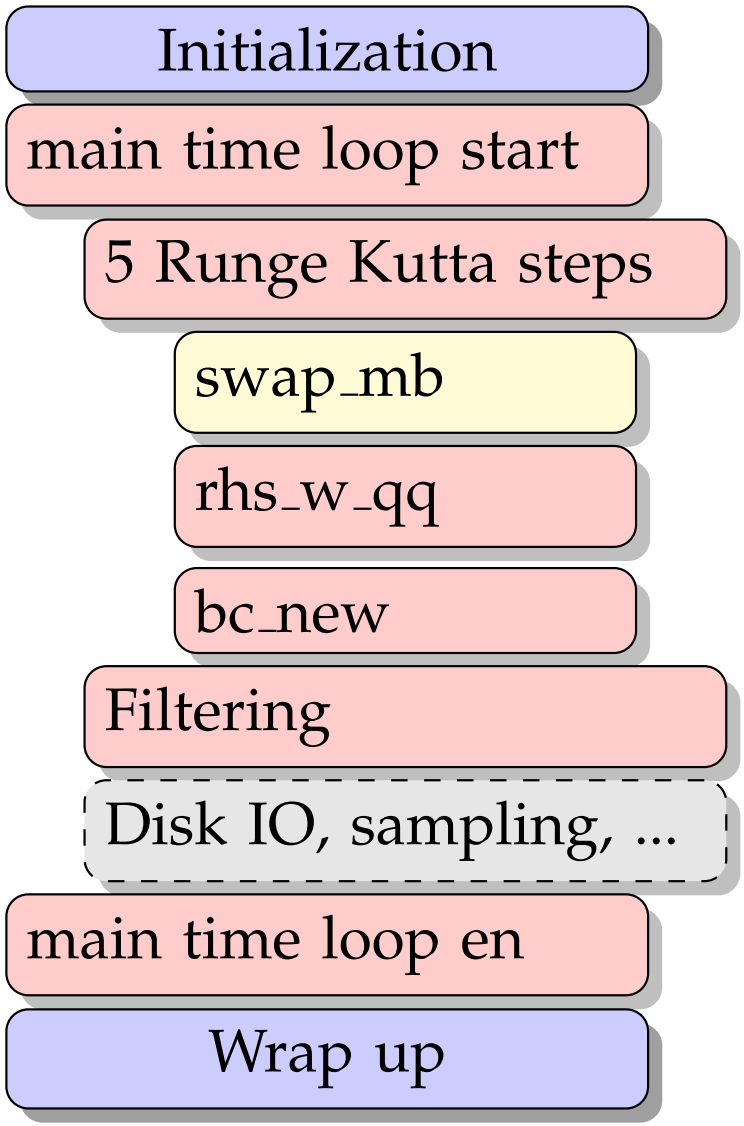
- 30% of walltime spent in CPU-GPU data transfer
- move everything to GPU except for
  - swapping - MPI communication
  - Disk IO - MPIIO

Step 2 - Limitting data transfer



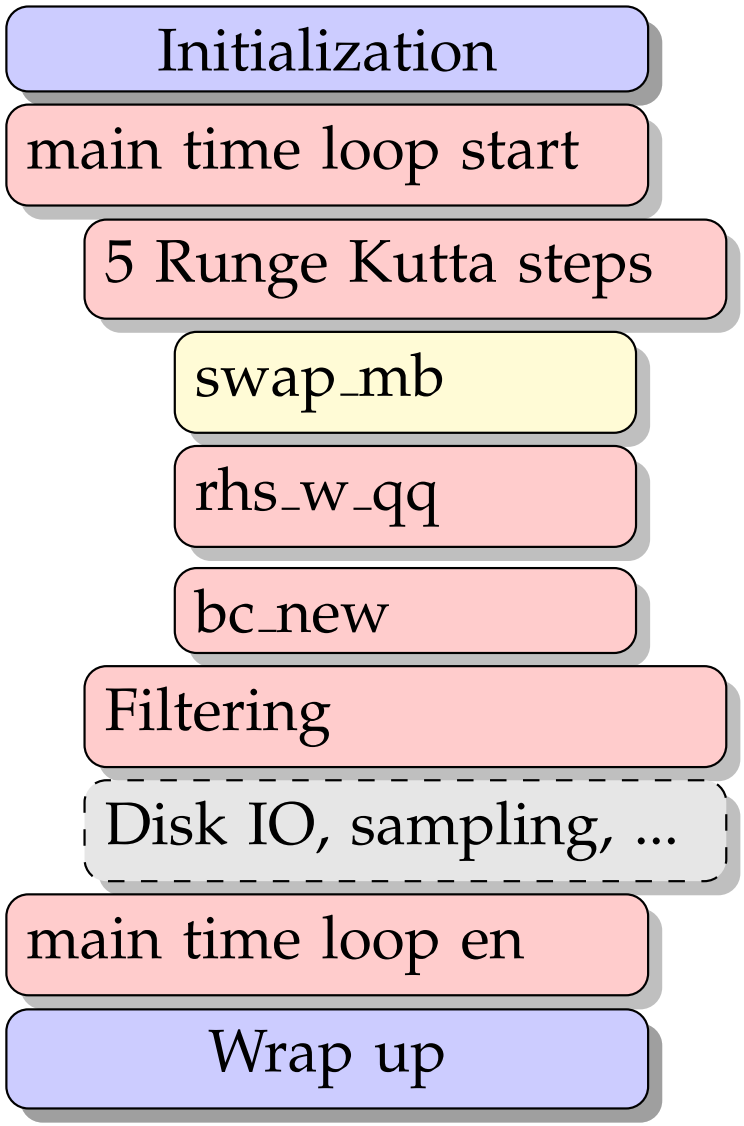
- 30% of walltime spent in CPU-GPU data transfer
- move everything to GPU except for
  - swapping - MPI communication
  - Disk IO - MPIIO
  - Initalization, Wrap up

Step 2 - Limitting data transfer



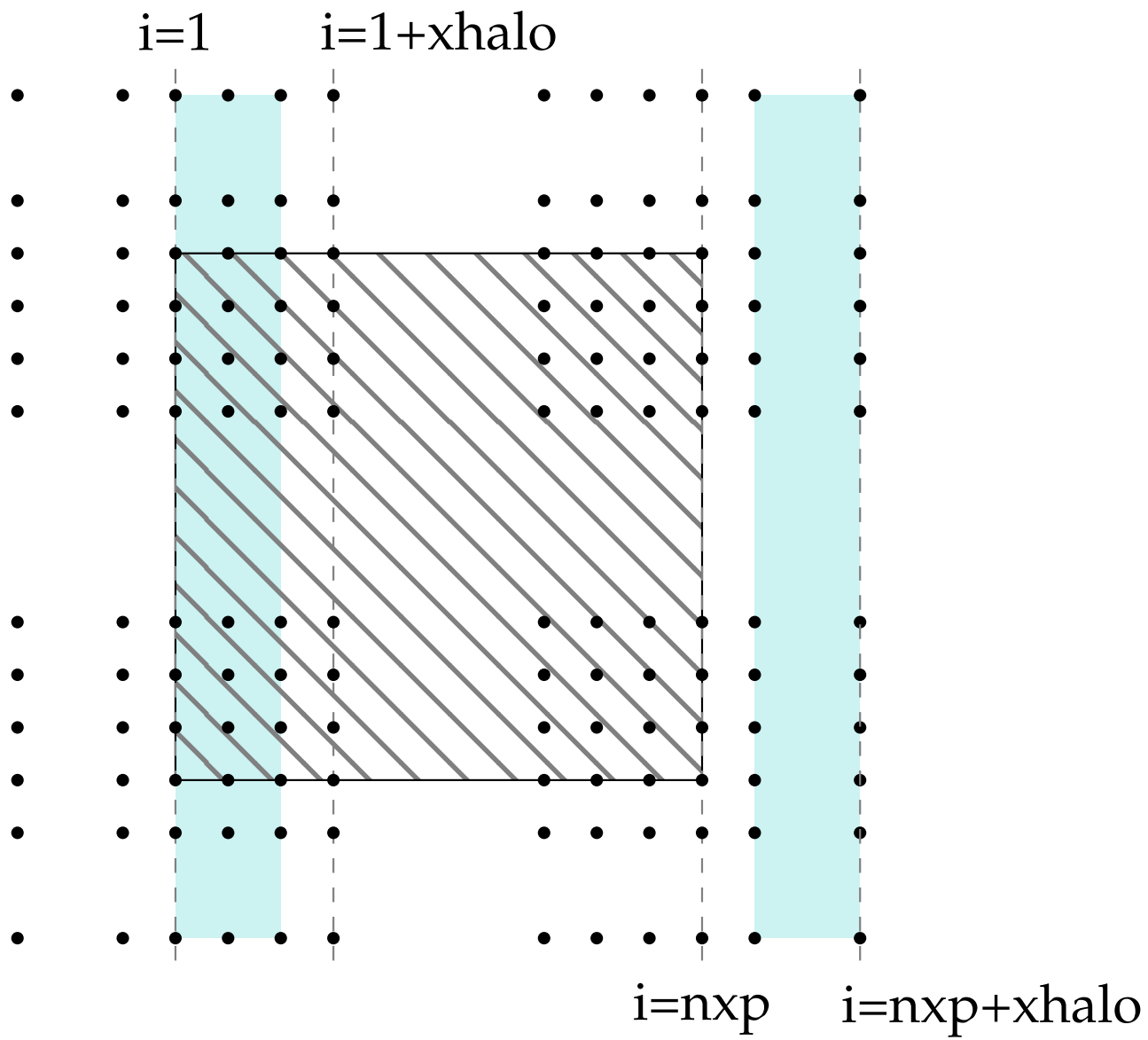
- 30% of walltime spent in CPU-GPU data transfer
- move everything to GPU except for
  - swapping - MPI communication
  - Disk IO - MPIIO
  - Initalization, Wrap up
- only halos are transfered between GPU and CPU
- about 70% of code have to be ported

Step 2 - Limitting data transfer



- 30% of walltime spent in CPU-GPU data transfer
- move everything to GPU except for
  - swapping - MPI communication
  - Disk IO - MPIIO
  - Initalization, Wrap up
- only halos are transfered between GPU and CPU
- about 70% of code have to be ported
- runtime increase by 7x

Step 2 - Limiting data transfer



STEP 2 - LIMITING DATA TRANSFER

```
!$acc parallel loop private(l)
        do n=1,nv
          do k=1,kpoints
           do j=1-yhalo,nyp+yhalo
             do i=1,xhalo
                l = i + (j+yhalo-1)*xhalo
     &            + (k-1)*(nyp+2*yhalo)*xhalo
     &            + (n-1)*(kpoints)*(nyp+2*yhalo)*xhalo
               ahoxm(l) = a(i,j,k,n)
             enddo
           enddo
         enddo
       enddo
```

- base version: $> 1600s$

Step 2 - Limiting data transfer

```
!$acc parallel loop private(l) collapse(2)
          do n=1,nv
            do k=1,kpoints
!$acc loop vector
              do j=1-yhalo,nyp+yhalo
!dir$ unroll
                do i=1,xhalo
                  l = i + (j+yhalo-1)*xhalo
     &              + (k-1)*(nyp+2*yhalo)*xhalo
     &              + (n-1)*(kpoints)*(nyp+2*yhalo)*xhalo
                  ahoxm(l) = a(i,j,k,n)
                enddo
              enddo
            enddo
          enddo
```

- base version: $> 1600s$
- final version: $\approx 12s$

STEP 2 - LIMITING DATA TRANSFER

```
!$acc parallel loop private(l) collapse(2)
          do n=1,nv
            do k=1,kpoints
!$acc loop vector
              do j=1-yhalo,nyp+yhalo
!dir$ unroll
                do i=1,xhalo
                  l = i + (j+yhalo-1)*xhalo
     &              + (k-1)*(nyp+2*yhalo)*xhalo
     &              + (n-1)*(kpoints)*(nyp+2*yhalo)*xhalo
                  ahoxm(l) = a(i,j,k,n)
                enddo
              enddo
            enddo
          enddo
```

- base version: $> 1600s$
- final version: $\approx 12s$
- $\approx 30 - 40\%$ speed up w.r.t. CPU version

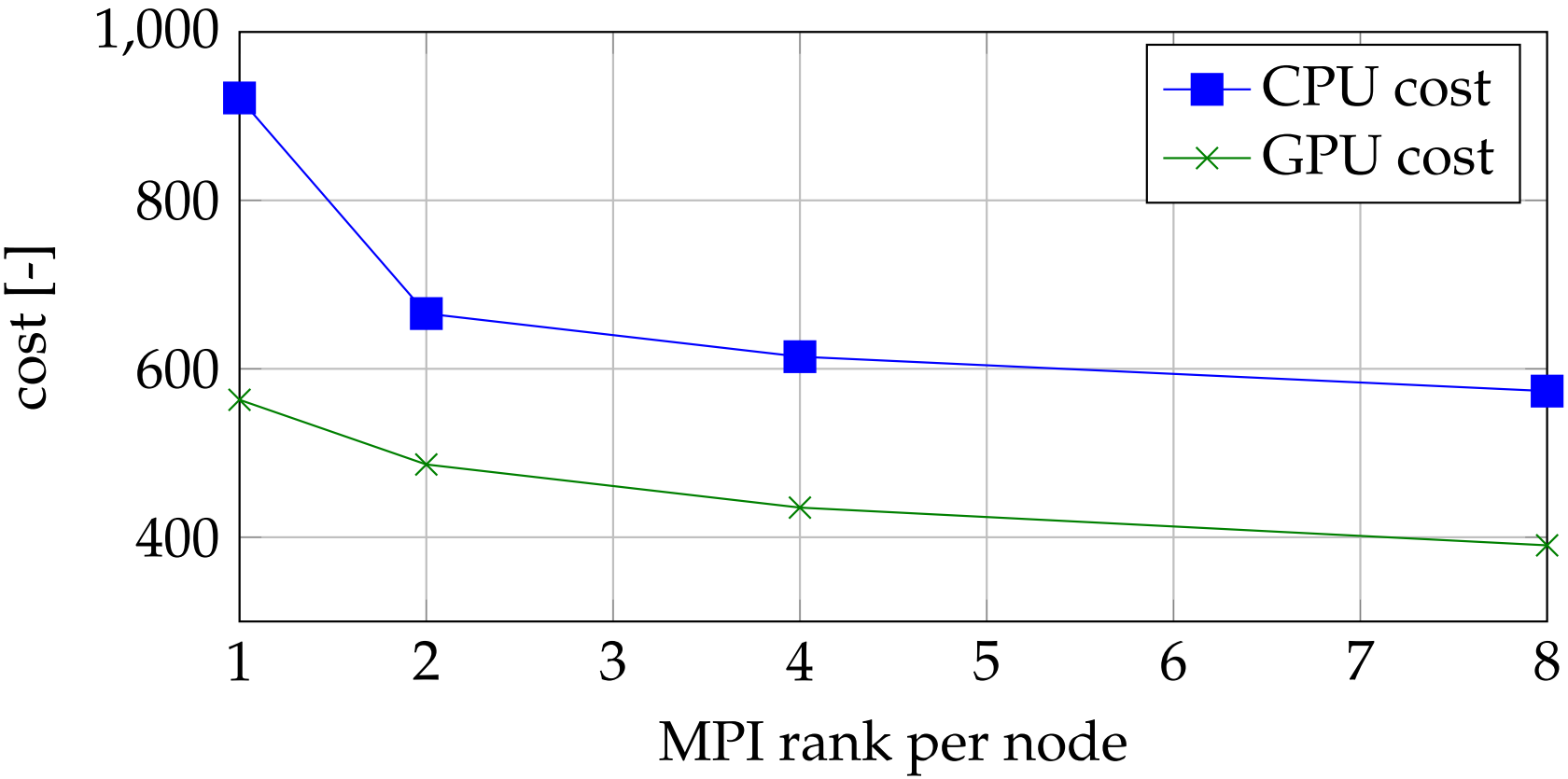
Step 2 - Limiting data transfer

```
!$acc parallel loop private(l) collapse(2)
          do n=1,nv
            do k=1,kpoints
!$acc loop vector
              do j=1-yhalo,nyp+yhalo
!dir$ unroll
                do i=1,xhalo
                  l = i + (j+yhalo-1)*xhalo
     &              + (k-1)*(nyp+2*yhalo)*xhalo
     &              + (n-1)*(kpoints)*(nyp+2*yhalo)*xhalo
                  ahoxm(l) = a(i,j,k,n)
                enddo
              enddo
            enddo
          enddo
```

- base version: $> 1600s$
- final version: $\approx 12s$
- $\approx 30 - 40\%$ speed up w.r.t. CPU version
- packing on CPU fastest if using l=l+1
- GPU faster as CPU

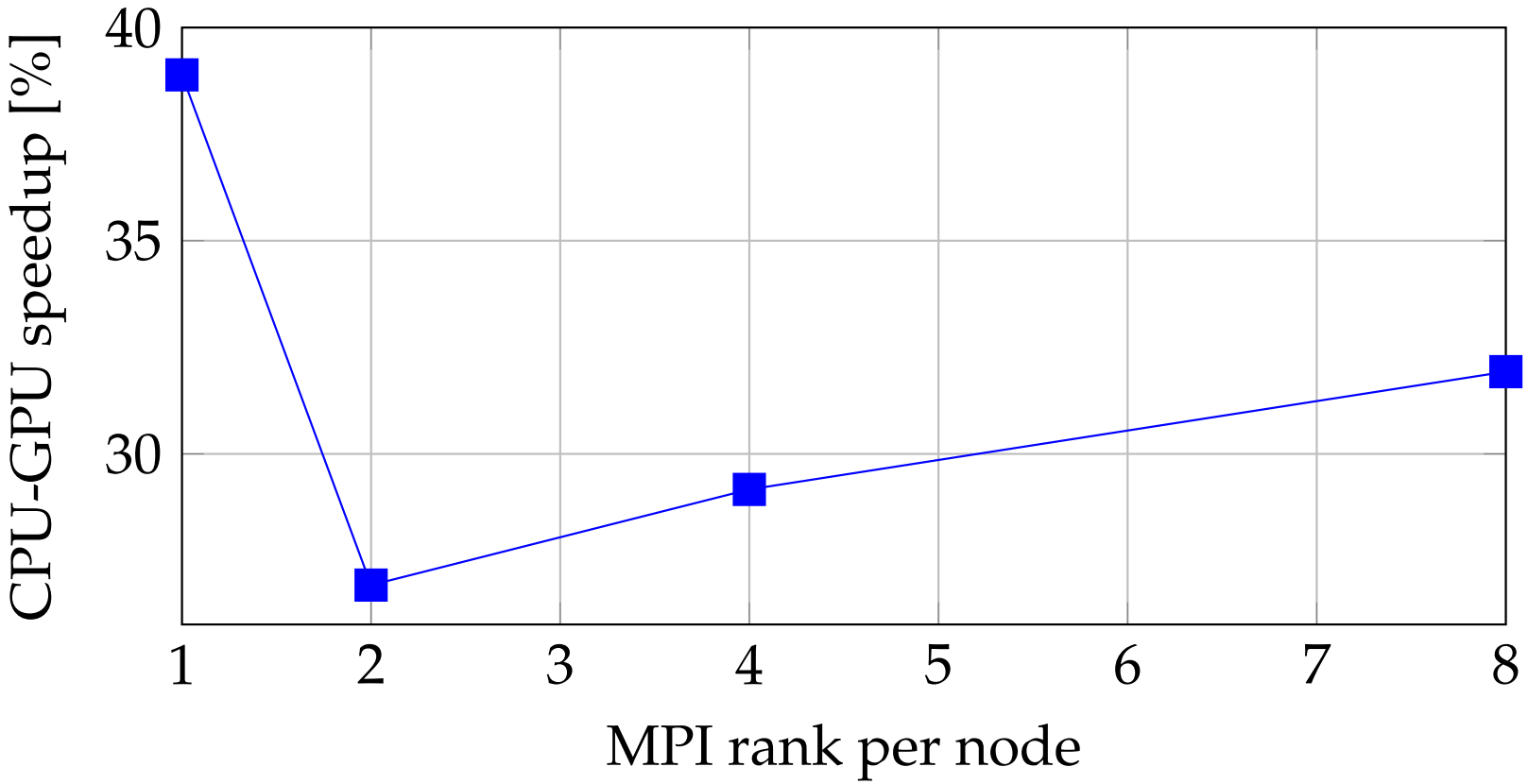
Step 2 - Speed-up

- strong scaling
- Using 64 - 512 nodes



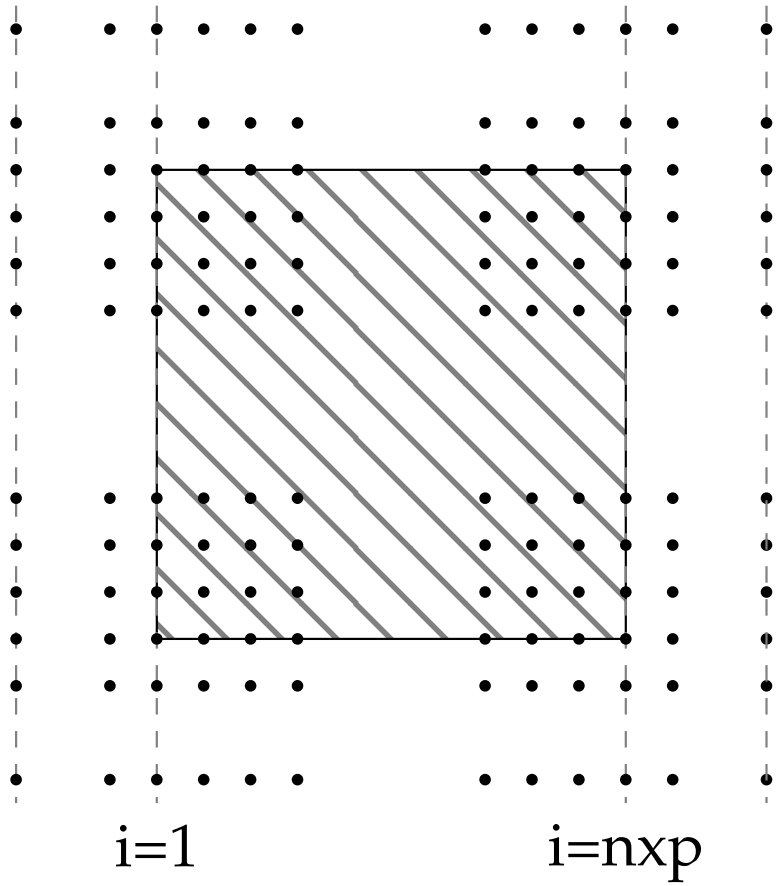
Step 2 - Speed-up

- strong scaling
- Using 64 - 512 nodes
- speed up around 30%



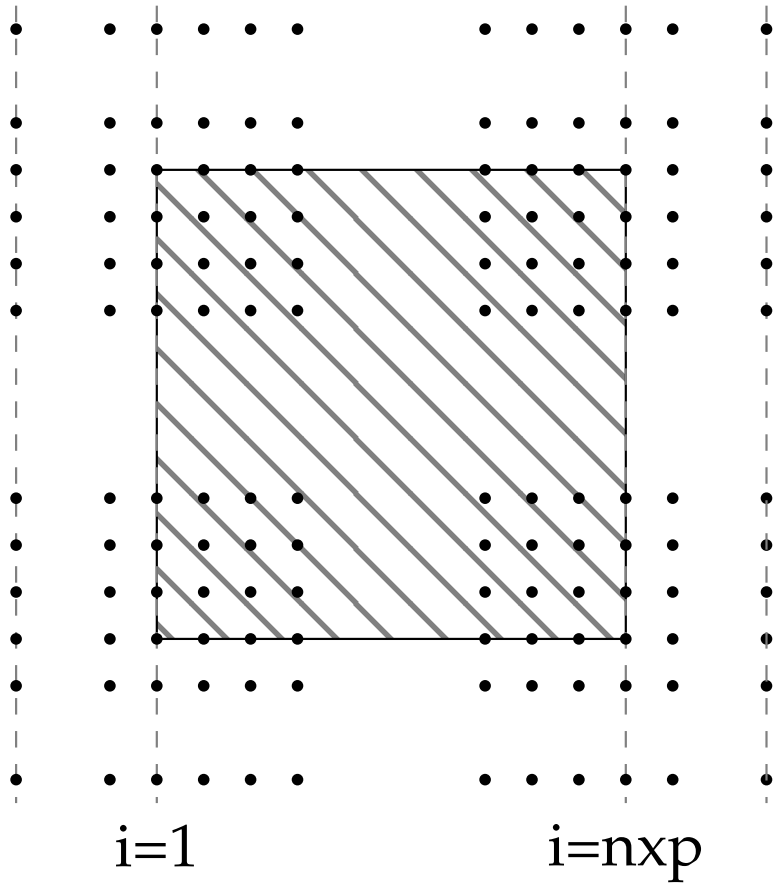
Step 3 - Additional tweaks

- mostly innermost loop length: nxp+2×xhalo
- generate grid such that innermost loop length is 32,64 or 128



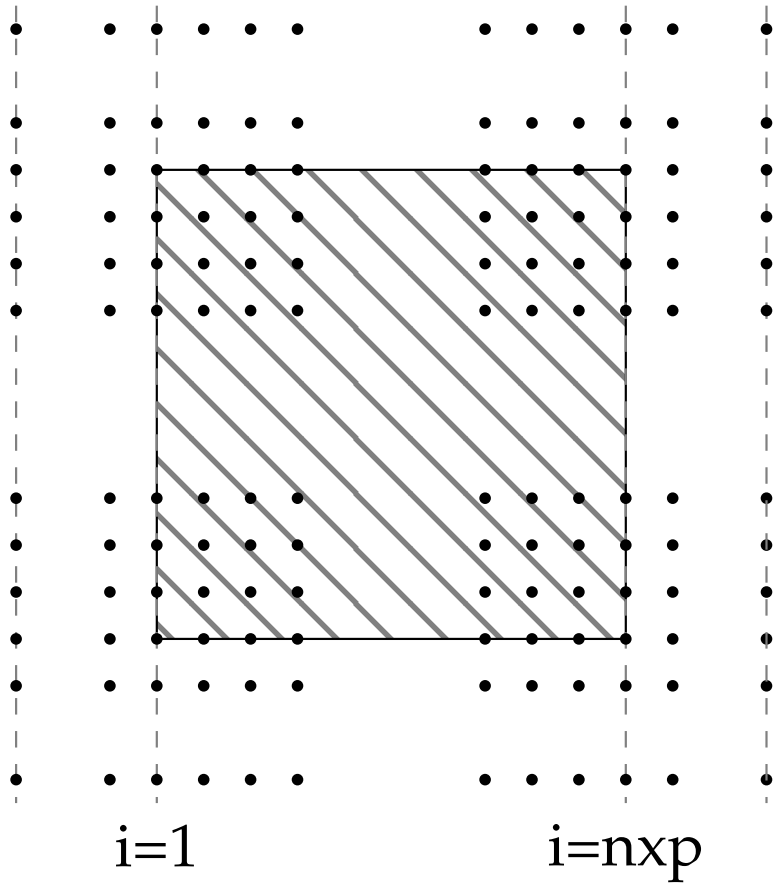
Step 3 - Additional tweaks

- mostly innermost loop length: nxp+2×xhalo
- generate grid such that innermost loop length is 32,64 or 128
- set vector_lenght() accordingly
- **set nxp,nyp,nzp as parameter**



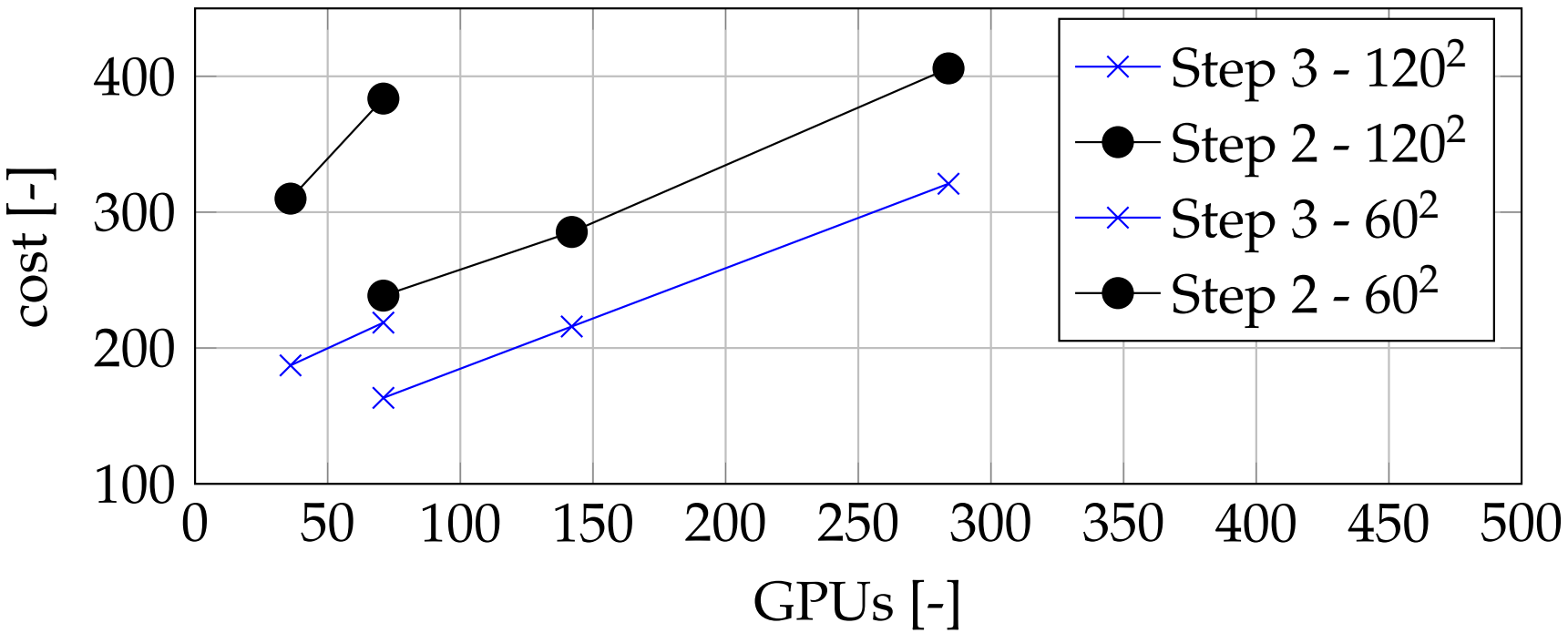
Step 3 - Additional tweaks

- mostly innermost loop length: nxp+2×xhalo
- generate grid such that innermost loop length is 32,64 or 128
- set vector_lenght() accordingly
- **set nxp,nyp,nzp as parameter**
- less flexible but significant performance improvement



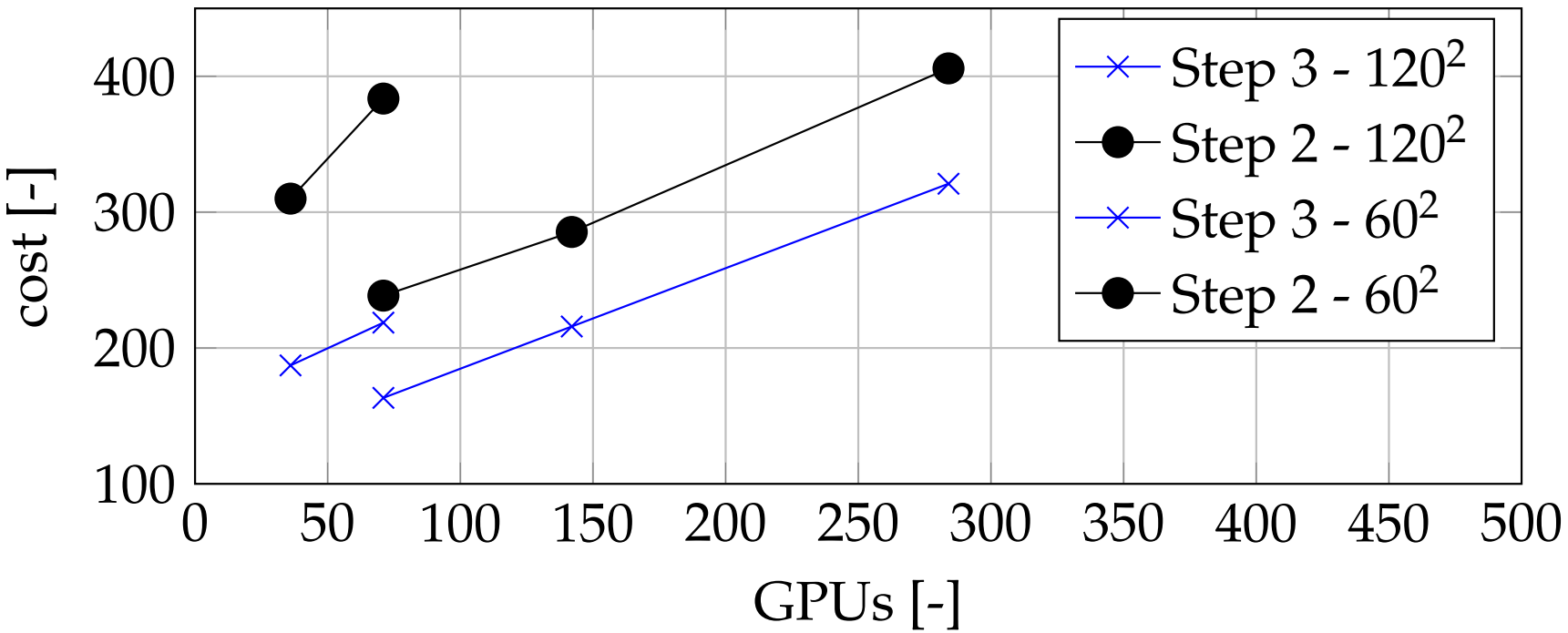
Step 3 - Speed up

- strong scaling
- almost 2x speed w.r.t. Step 2



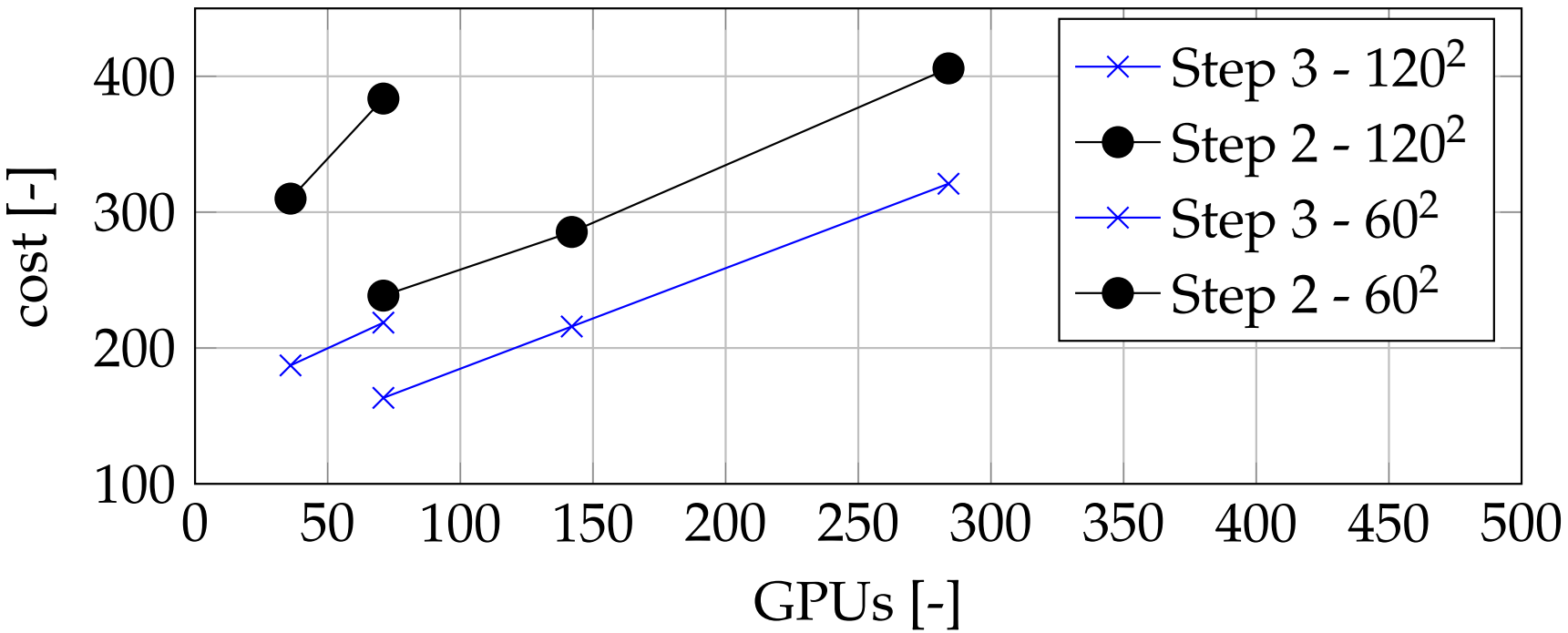
Step 3 - Speed up

- strong scaling
- almost 2x speed w.r.t. Step 2
- “best performance” by placing 4 MPI ranks
- try 28^2 (vector_length(32))



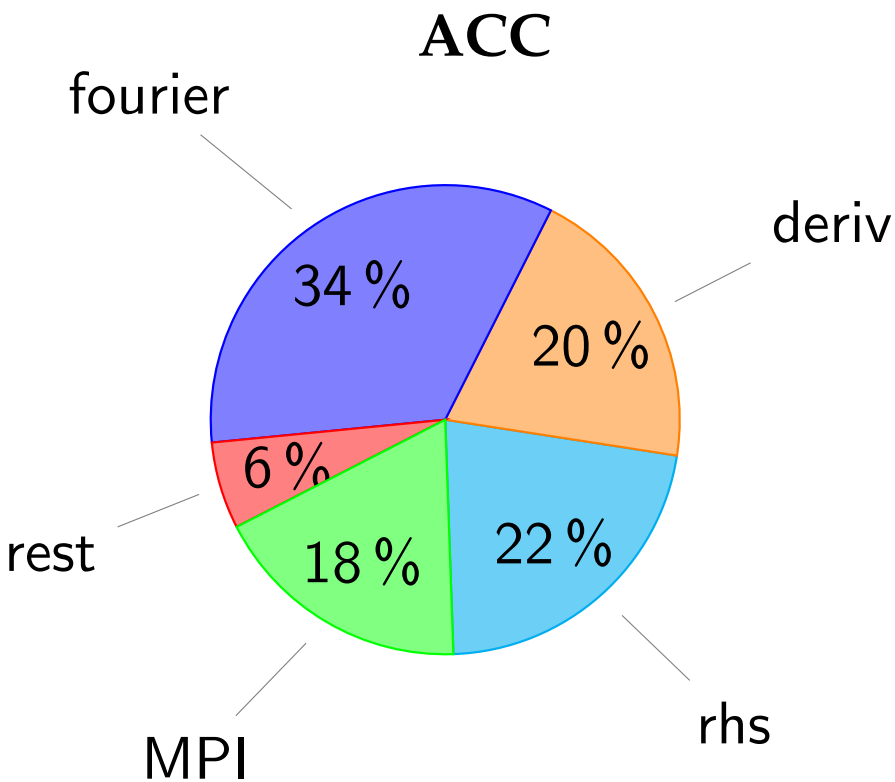
Step 3 - Speed up

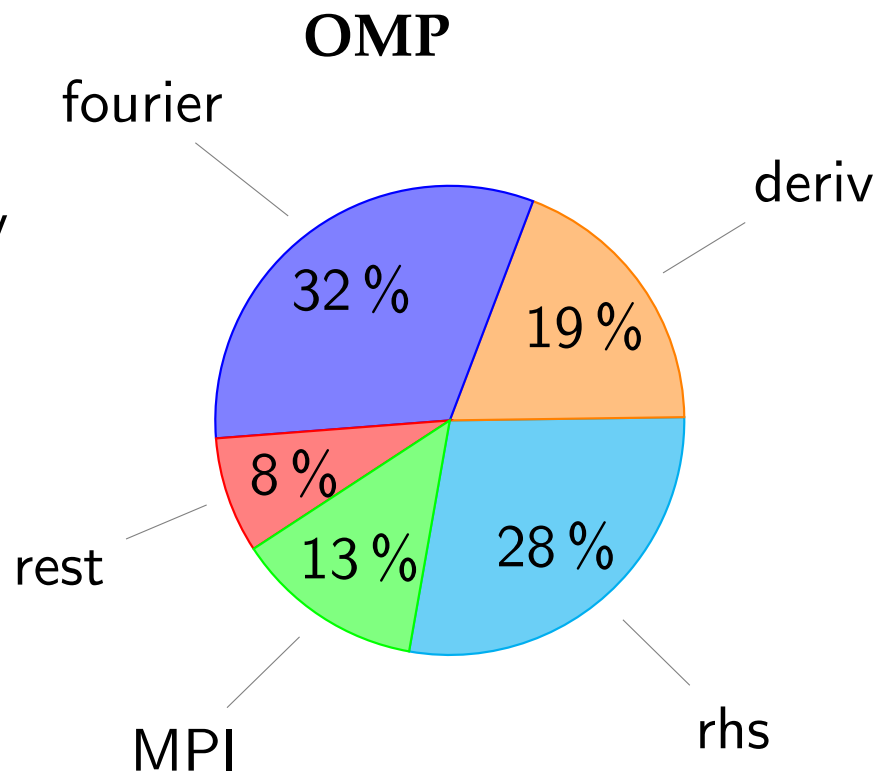
- strong scaling
- almost 2x speed w.r.t. Step 2
- “best performance” by placing 4 MPI ranks
- try 28^2 (vector_length(32))
- speed up w.r.t. CPU $\approx$ 2x



Step 3 - Profiles

- Piz Daint
- CrayPAT only works for 1 MPI rank per node
- similar distribution of work according to routines





Conclusions and Future work

- HiPSTAR achieved more than 2x speed up (1 GPU vs. 1 CPU)
- GPU code faster but less flexible

Conclusions and Future work

- HiPSTAR achieved more than 2x speed up (1 GPU vs. 1 CPU)
- GPU code faster but less flexible
- CPU does not perform number crunching $\Rightarrow$ different ratio of CPU to GPU

Conclusions and Future work

- HiPSTAR achieved more than 2x speed up (1 GPU vs. 1 CPU)
- GPU code faster but less flexible
- CPU does not perform number crunching $\Rightarrow$ different ratio of CPU to GPU

ACC Hackaton CSCS

- correct problems with PGI compilers
- port rest of the code: FSI, Adjoint method
- cooperate with experts

Conclusions and Future work

- HiPSTAR achieved more than 2x speed up (1 GPU vs. 1 CPU)
- GPU code faster but less flexible
- CPU does not perform number crunching $\Rightarrow$ different ratio of CPU to GPU

ACC Hackaton CSCS

- correct problems with PGI compilers
- port rest of the code: FSI, Adjoint method
- cooperate with experts

Longer term (Summit)

- Multi GPU environment - which route to go?
- Are the algorithms optimized for CPU as well the best ones for GPU?