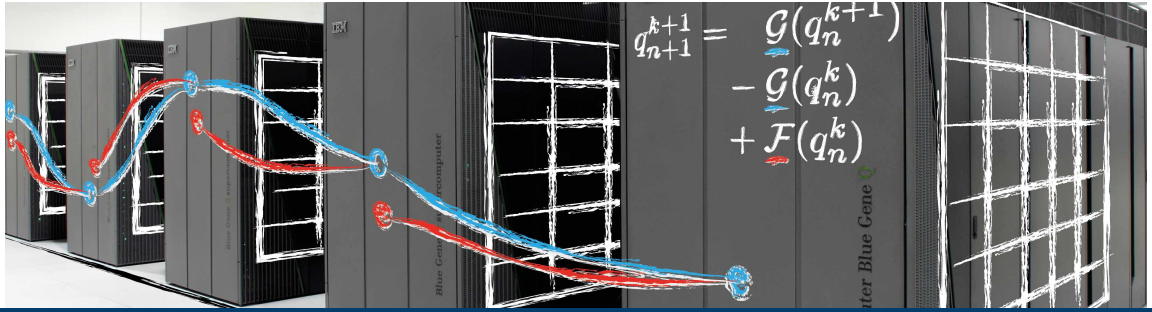


Massively parallel adaptive spectral deferred correction in Python

December 9, 2024 | Thomas Baumann | Jülich Supercomputing Centre

Agenda

Introduction to SDC

Adaptivity in SDC
- Boosting computational efficiency
- Boosting resilience

Python on HPC
- Python on GPUs
- Parallel scaling
- Large space-time parallel simulation

Spectral deferred correction (SDC)[1]

Goal

Solve initial value problem (IVP): $u_t = f(u), \quad u(0) = u_0$

SDC idea

- Integrate IVP w.r.t. time: $u(t) = u_0 + \int_0^t f(u(s))ds$
- Compute integral over interval $(0,\Delta t]$ numerically with quadrature rule Q
  $\rightarrow$ defines a fully implicit Runge-Kutta method (RKM)
- Solve fully implicit RKM iteratively using preconditioned Picard iteration

[1] [Dutt et al., 2000]

SDC: Numerical integration with quadrature rule

Compute quadrature rule

- Divide the interval into M quadrature nodes $0 \leq \tau_m \leq \Delta t$, with $u_m \approx u(\tau_m)$
- Compute interpolating Lagrange polynomials: $l_j^\tau(t)$ with $l_j^\tau(\tau_i) = \delta_{ij}$
  $\rightarrow$ Use for polynomial approximation of $f(t) = \sum_{m=1}^{M} l_m^\tau(t) f(u_m)$
- Integrate polynomial approximation:
  $u_m = u_0 + \int_0^{\tau_m} \sum_{j=1}^{M} l_j^\tau(t) f(u_j) dt = u_0 + \sum_{j=1}^{M} f(u_j) \int_0^{\tau_m} l_j^\tau(t) dt = u_0 + \sum_{j=1}^{M} q_{mj} f(u_j)$

Collocation problem: Exact integral of order M polynomial approximation

$$\boxed{u_m = u_0 + \sum_{j=1}^{M} q_{mj} f(u_j), \quad m = 1,...,M} \iff \vec{u} = \vec{u}_0 + Qf(\vec{u})$$

Collocation problem

Pros

- Polynomial approximation is order M accurate anywhere within $(0, \Delta t]$
- For spectral quadrature rules: Get up to order $2M$ at Δt

Cons

- $Q \in \mathbb{R}^{M \times M}$ is dense
- For PDE with N degrees of freedom, need to solve $MN \times MN$ systems in collocation problem

SDC to the rescue

Solve collocation problem iteratively with forward substitution $\rightsquigarrow$ solve kM-many $N \times N$ systems

SDC iteration

Recall: Want to solve $\vec{u} = \vec{u}_0 + Qf(\vec{u})$

Simplest iterative scheme: Picard iteration

$$\vec{u}^{k+1} = \vec{u}^k + \vec{r}^k, \quad \vec{r}^k := \vec{u}_0 + \Delta t Q F(\vec{u}^k) - \vec{u}^k$$

- Gain one order of accuracy per iteration up to the order of Q
- Method is explicit $\rightarrow$ poor stability

Use preconditioner Q_Δ

$$(I_M - \Delta t Q_\Delta F)\left(\vec{u}^{k+1}\right) = \vec{u}_0 + \Delta t \left(Q - Q_\Delta\right) F\left(\vec{u}^k\right)$$

- Still gain one order of accuracy per iteration up to the order of Q (most of the time... [Causley and Seal, 2019, Christlieb et al., 2009])
- Q_Δ is lower triangular $\rightsquigarrow$ forward substitution

SDC preconditioners

Easy splitting

$$\left(I_M - \Delta t \sum_i Q_\Delta^i F\right)\left(\vec{u}^{k+1}\right) = \vec{u}_0 + \Delta t \left(Q - \sum_i Q_\Delta^i\right) F\left(\vec{u}^k\right)$$

$\rightarrow$ We will later use IMEX splitting by using one preconditioner that is strictly lower triangular and one with non-zero diagonal entries [Ruprecht and Speck, 2016]

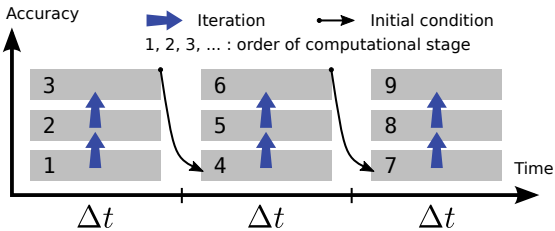
Small scale time-parallelism via diagonal Q_Δ [van der Houwen and Sommeijer, 1991]

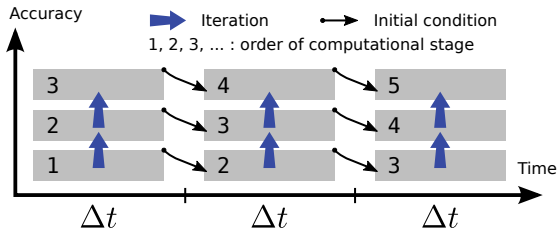
- Update the nodes in parallel on M tasks
- Very good preconditioners where recently published by Gaya and Thibaut [Čaklović et al., 2024]

Optimal preconditioner is problem dependent, not known a priori and an active area of research

Parallel-in-time extension: Block Gauß-Seidel SDC

[Guibert and Tromeur-Dervout, 2007]





Start iteration on step as soon as one iteration has been performed on previous step

Figures by Thibaut Lunet

Adaptive step size selection in SDC

- SDC has a lot in common with RKM
- Step size adaptivity is mature in RKM with embedded methods

$\rightarrow$ transfer concepts from embedded RKM to SDC

Embedded Runge-Kutta methods

- Idea: Control step size to match local error to target tolerance ϵ_{TOL}
- Error estimate: subtract solutions of different orders p and q, $q > p$:

$$\epsilon = \|u^{(p)} - u^{(q)}\| = e^{(p)} + \mathcal{O}(\Delta t^{q+1})$$

  $\rightarrow$ This estimates local error of lower order method
- Step size update: Infer "optimal" step size via the order ($\beta = 0.9$: safety factor)

$$\Delta t_{\mathrm{opt}} = \beta \left(\frac{\epsilon_{\mathrm{TOL}}}{\epsilon} \right)^{1/(p+1)}$$

- Restart current step if $\epsilon > \epsilon_{\mathrm{TOL}}$
- Obtain lower order methods from same stages but different weights as higher order method

Step size adaptivity in SDC

Transfer ideas from embedded RKM

- Use same step size update equation
- Use bespoke error estimates

Resulting algorithms

- Δt-adaptivity: Constant number of iterations, adaptive step size
- Δt-k-adaptivity: Choose both adaptively

Δt-adaptivity

Error estimate

Order of SDC after k iterations: k
(conditions apply)

$$\epsilon = \|u^{k-1} - u^k\|$$

Step size update

$$\Delta t_{\text{opt}} = \beta \left(\frac{\epsilon_{\text{TOL}}}{\epsilon}\right)^{1/k}$$

Δt-k-adaptivity

- Recall: Solution to collocation problem is order M polynomial approximation in $(0,\Delta t]$ by u_m, $m = 0,...,M$
- Idea: Construct secondary order $M-1$ solution via polynomial interpolation at τ_{M-1}
  - Select nodes $\tau^* = \{\tau_i, i = 0,1,...,M-2,M\}$
  - Compute interpolation weights via Lagrange polynomials $w_j = l_j^{\tau^*}(t = \tau_{M-1})$

Error estimate

$$\epsilon = \| \sum_{m=0}^{M-2} w_j u_j + w_{M-1} u_M - u_{M-1} \|_\infty$$

Step size update

$$\Delta t_{\text{opt}} = \beta \left(\frac{\epsilon_{\text{TOL}}}{\epsilon} \right)^{1/M}$$

Step size adaptivity for diagonal SDC

Diagonal SDC

Use diagonal Q_Δ, everything else is the same

Δt-adaptivity

Numerically verify you get the expected order after every iteration

Δt-k-adaptivity

Business as usual

Step size adaptivity for Block Gauß-Seidel SDC

Block Gauß-Seidel SDC

Solve entire block with first order method, then solve entire block with second order method, ...

Δt-adaptivity

- Increment on last step in block behaves like global error within the block
  $\rightarrow$ Make sure error due to parallelization does not get out of hand
- Overestimates local error a bit, but can be used as usual

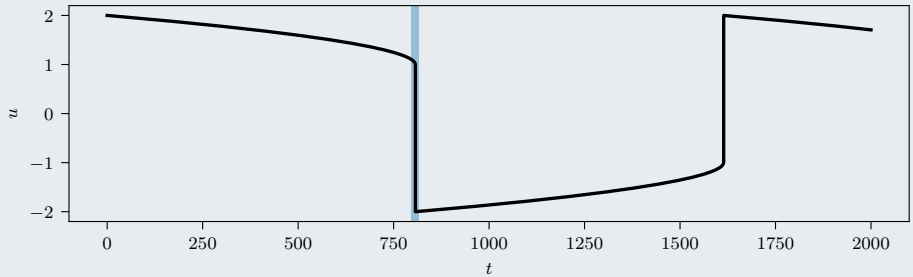
Δt-k-adaptivity

Business as usual

Benchmark problems

Van der Pol Oscillator

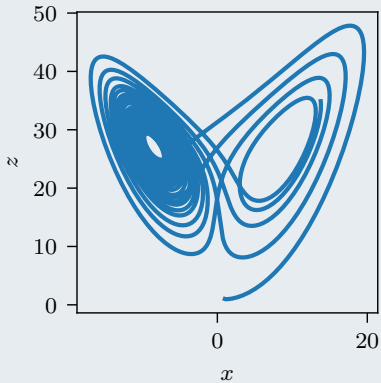
$$u_{tt} - \mu \left(1 - u^2\right) u_t + u = 0$$

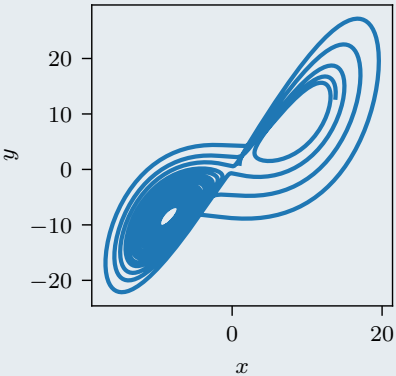


Benchmark problems

Lorenz attractor

$$x_t = \sigma(y - x)$$
$$y_t = \rho x - y - xz$$
$$z_t = xy - \beta z$$



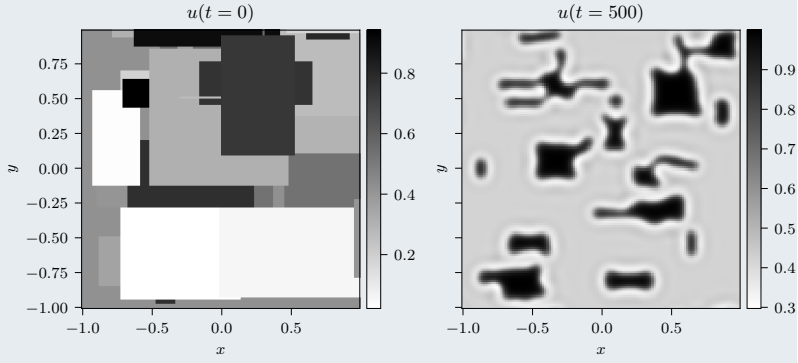


Benchmark problems

Gray-Scott

$$u_t = \nu_u \Delta u - uv^2 + F(1-u)$$
$$v_t = \nu_v \Delta v + uv^2 - (F+k)v$$

Solve IMEX with Fourier pseudo-spectral discretization



Benchmark problems

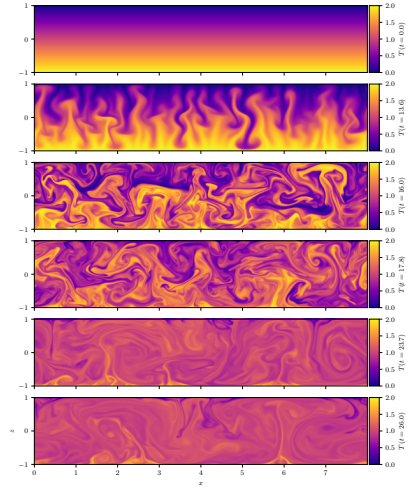
Rayleigh-Benard convection

Incompressible fluid dynamics:

$$
\begin{aligned}
u_t - \nu(u_{xx} + u_{zz}) + p_x &= -uu_x - vu_z \\
v_t - \nu(v_{xx} + v_{zz}) + p_z - T &= -uv_x - vv_z \\
T_t - \kappa(T_{xx} + T_{zz}) &= -uT_x - vT_z \\
u_x + v_z &= 0
\end{aligned}
$$

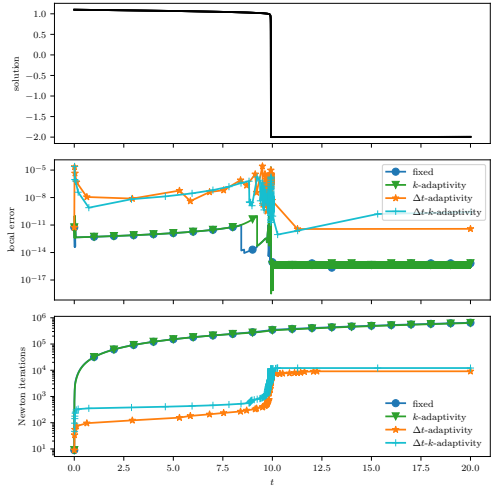
Space discretization

- Solve IMEX with Fourier + ultraspherical pseudo-spectral discretization
- Differential algebraic equation $\rightarrow$ need stiffly accurate integrator



JÜLICH Forschungszentrum

Boosting computational efficiency with adaptivity



SDC strategies

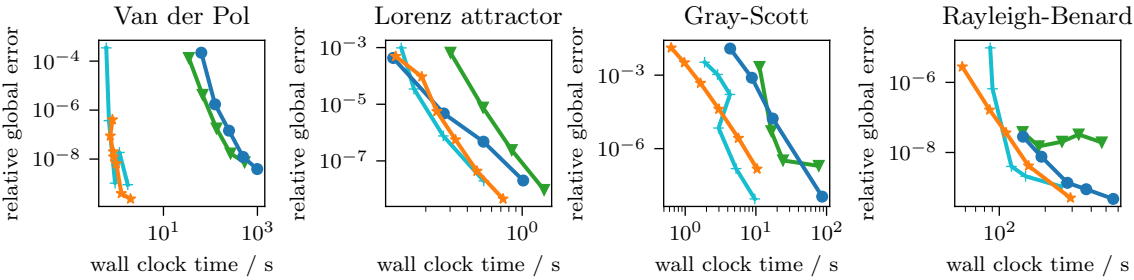
- fixed: fixed iteration number k and step size Δt
- k-adaptivity: adaptive k, fixed Δt
- Δt-adaptivity: adaptive Δt, fixed k
- Δt-k-adaptivity both adaptive

Results

- k-adaptivity: Can't tune k finely enough
- Need adaptive Δt to accommodate changes in timescale

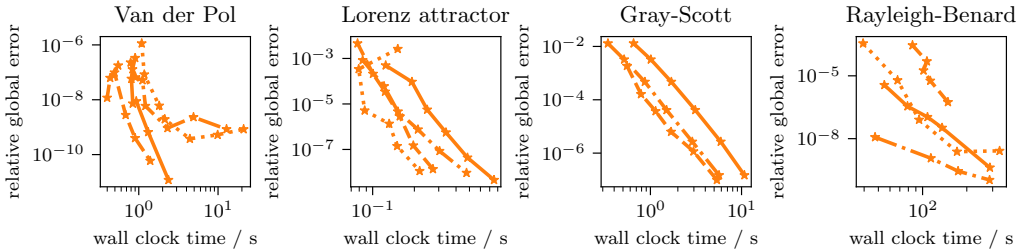
JÜLICH Forschungszentrum

Boosting computational efficiency with adaptivity



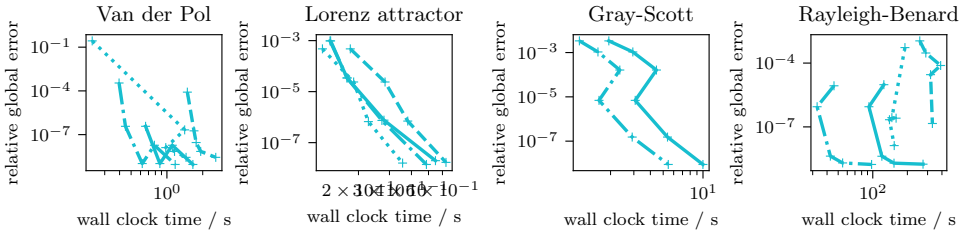
- k-adaptivity: Computing residual is expensive for ODEs
- Δt-adaptivity works well for any accuracy and problem
- Δt-k-adaptivity works best for tight tolerance

Parallel-in-time Δt-adaptivity



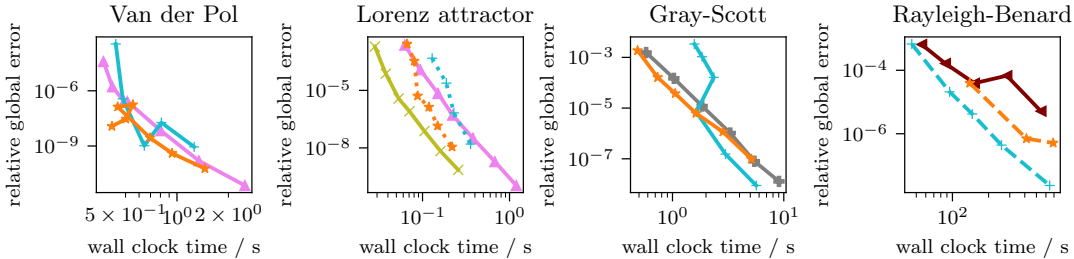
- Performance heavily depends on Q_Δ
- Don't know beforehand if speedup is possible
- At least one parallelization method always gives good speedup
- Combining diagonal SDC and block Gauß-Seidel SDC is seldom worthwhile

Parallel-in-time Δt-k-adaptivity



- Similar poperties w.r.t. parallelisation as Δt-adaptivity
- Here, we often use diagonal Q_Δ in serial as well
  $\rightarrow$ Get same result in diagonal SDC
- Particularly good speedup with diagonal SDC for PDEs

SDC vs. RKM



- At least mode of adaptive PinT SDC is always competitive with RKM for stiff problems
- In Rayleigh-Benard, no high order comparison RKM available, SDC still better at order 3

Resilience

Tyranny of numbers

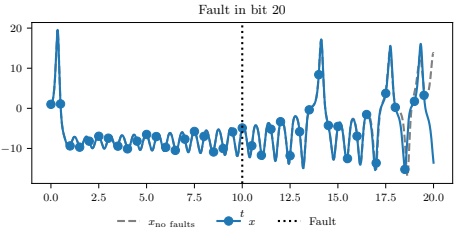
- Insane number of components
- Large chance of failure somewhere
- E.g.: Mariner 8 crashed because of a single faulty diode

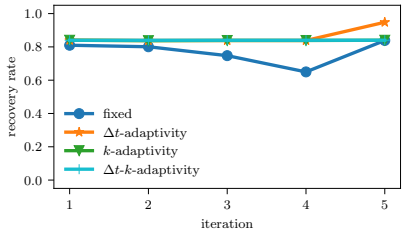
Faults in modern HPC

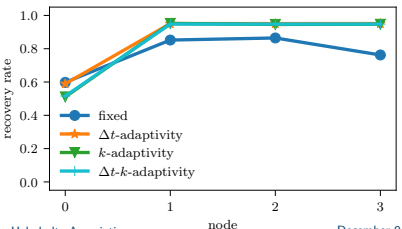
- Sources: Radiation and hardware damage
- Multiple times each day in HPC machines [Glosli et al., 2007, Schroeder et al., 2011]

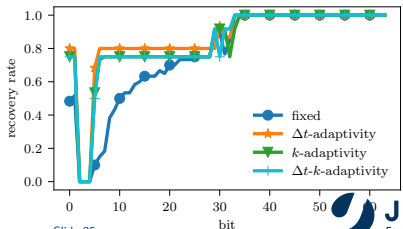
Boosting resilience by adaptivity

Lorenz attractor



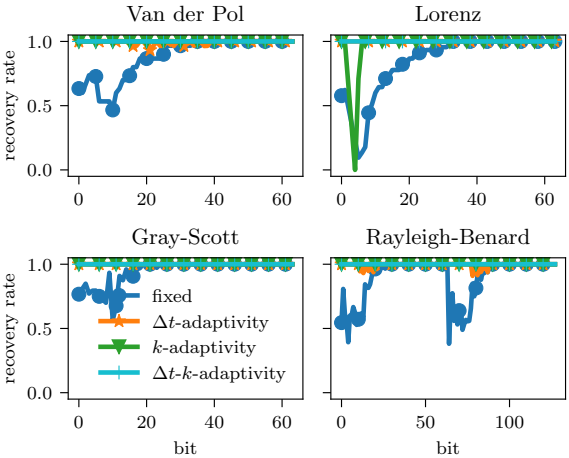






JÜLICH Forschungszentrum

Boosting resilience by adaptivity



- Note: We exclude faults to initial conditions or ones that cause crashes here
- k-adaptivity: Iteration may not converge from faulty initial guess
- Δt-adaptivity: Some faults to intermediate bits don't trigger restart, but show up in the end
- Adaptive step size selection greatly improves resilience!

GPU programming

Analogy: Logging

- CPU: Tiny fast boat arranging logs in stream
- GPU: River that's doing the heavy lifting

Actual programming

- CPU submits operations ("kernels") to stream(s) on GPU
- GPU asynchronously executes operations
- Need to synchronize explicitly if CPU interacts with data on GPU

Porting Python code to GPU

Python philosophy

Perform actual computations within libraries that wrap compiled code

CPU libraries

- NumPy
- mpi4py
- FFTW
- SuperLU

GPU libraries

- CuPy
- NCCL (ships with CuPy)
- cuFFT (ships with CuPy)
- *Hier könnte ihr GPU sparse solver stehen!*

Porting Python code to GPU is mostly easy!

Note that MPI is not stream aware!

Porting Gray-Scott implementation to GPU

Fourier pseudo-spectral method

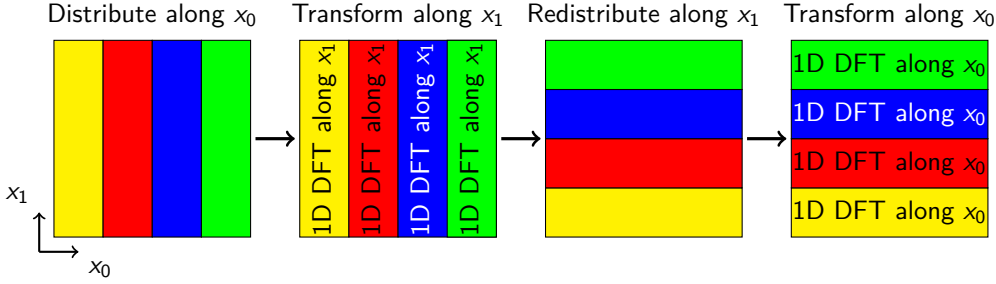
- Laplacian is diagonal $\rightarrow$ invert by division
- Right hand side evaluations: Need only multiplication and addition
- Need library for distributed FFTs! $\rightarrow$ mpi4py-fft

GPU porting

- Swap NumPy for CuPy
- Port mpi4py-fft to GPUs

Port mpi4py-fft to GPU

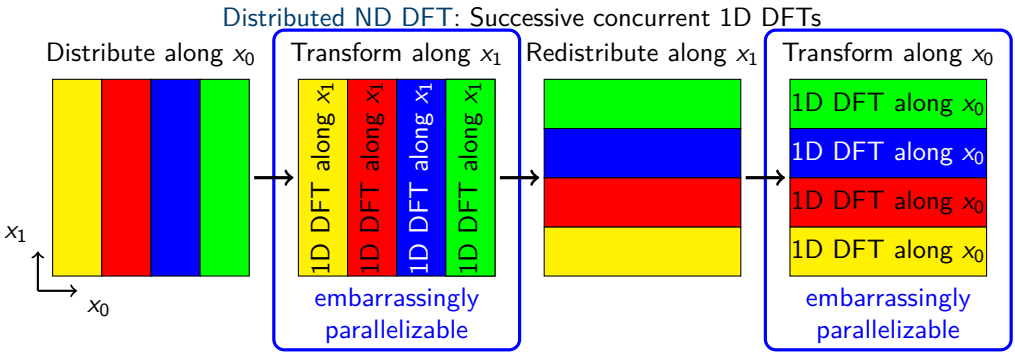
Distributed ND DFT: Successive concurrent 1D DFTs



Port to GPU

- Implement CuPy array interface
- Replace FFTW with cuFFT
- Implement Alltoall in NCCL
- Add NCCL as communication back-end

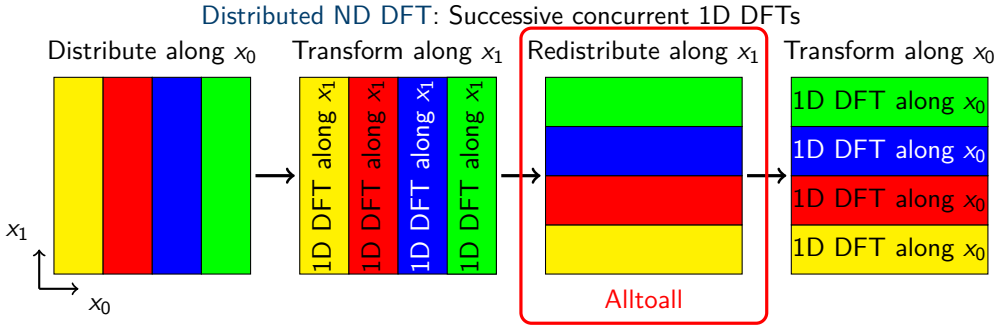
Port mpi4py-fft to GPU



Port to GPU

- Implement CuPy array interface
- Replace FFTW with cuFFT
- Implement Alltoall in NCCL
- Add NCCL as communication back-end

Port mpi4py-fft to GPU

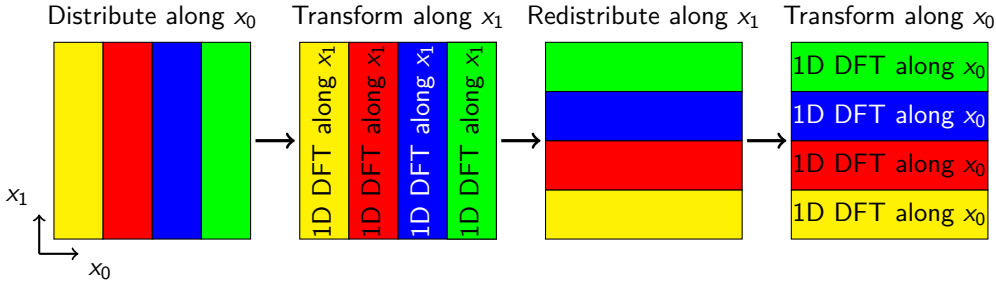


Port to GPU

- Implement CuPy array interface
- Replace FFTW with cuFFT
- Implement Alltoall in NCCL
- Add NCCL as communication back-end

Port mpi4py-fft to GPU

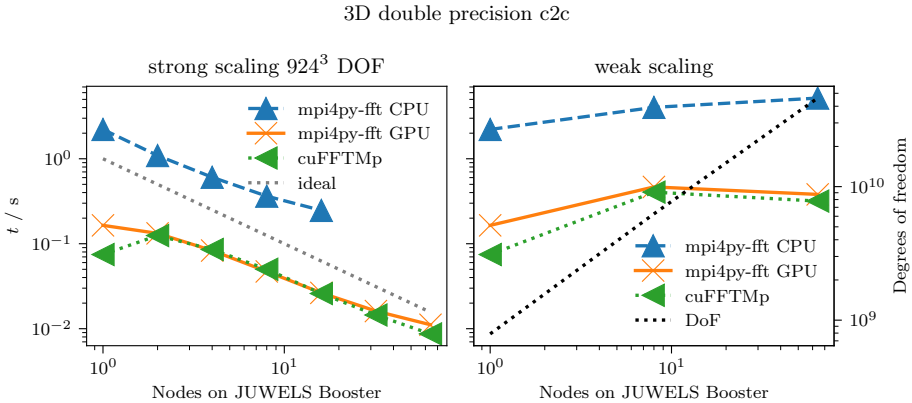
Distributed ND DFT: Successive concurrent 1D DFTs



Port to GPU

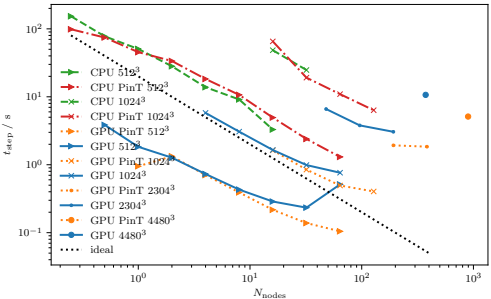
- Implement CuPy array interface
- Replace FFTW with cuFFT
- Implement Alltoall in NCCL
- Add NCCL as communication back-end

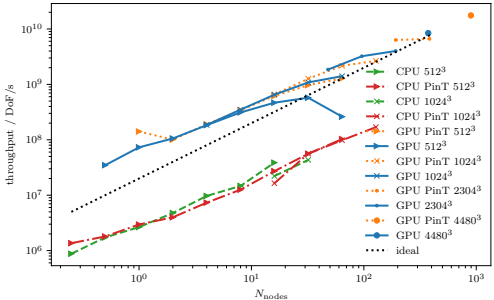
Parallel scaling of mpi4py-fft GPU port



Remarkably competitive with optimized NVIDIA competition cuFFTMp

Parallel scaling of Gray-Scott implementation





GPUs significantly outperform CPUs

- 16 tasks per node on CPU cluster
- 4 tasks per node on GPU cluster
- $\approx$ 10x speedup at same number of nodes

Use diagonal SDC to extend scaling

- Shifts from all-to-all to reduce
- Improves strong scaling
- Enables scaling up to 3584 GPUs

Porting RBC implementation to GPU

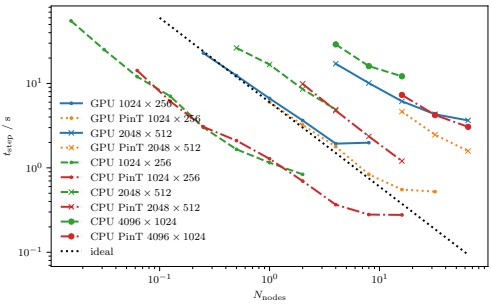
Fourier and ultraspherical pseudo-spectral method

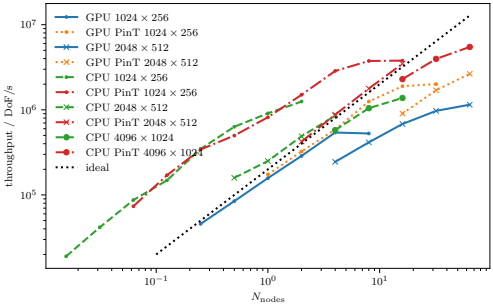
- System matrix is non-diagonal $\rightarrow$ solve by LU decomposition using SuperLU
- Right hand side evaluations: Need only multiplication and addition
- Need library for distributed FFTs and DCTs $\rightarrow$ mpi4py-fft

GPU porting

- Swap NumPy for CuPy
- Use port of mpi4py-fft to GPUs
- ↯No suitable replacement for SuperLU available at the moment $\rightarrow$ LU decompositions stay on CPU

Parallel scaling of RBC implementation





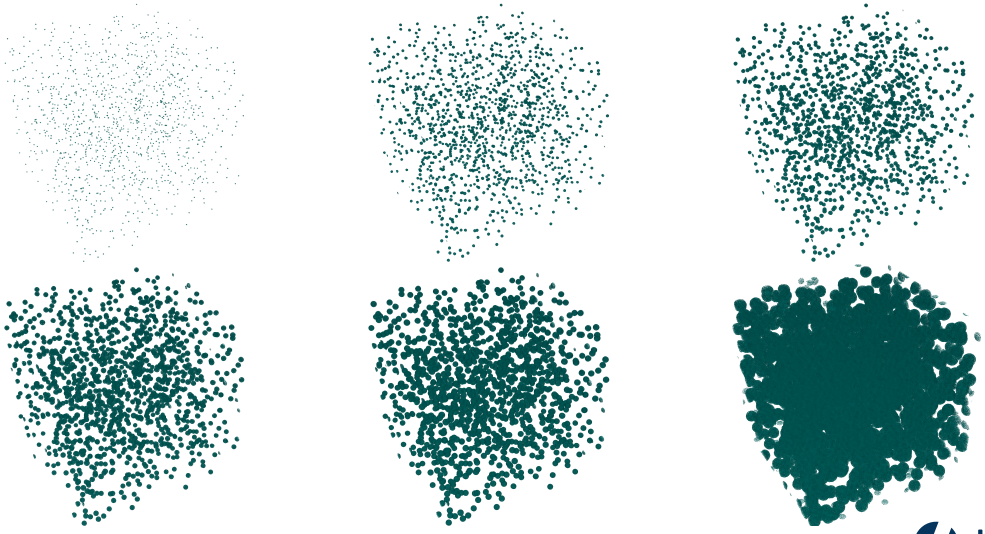
CPUs outperform GPUs

- 64 tasks per node on CPU cluster
- 4 tasks per node on GPU cluster
- LU takes most of the time!

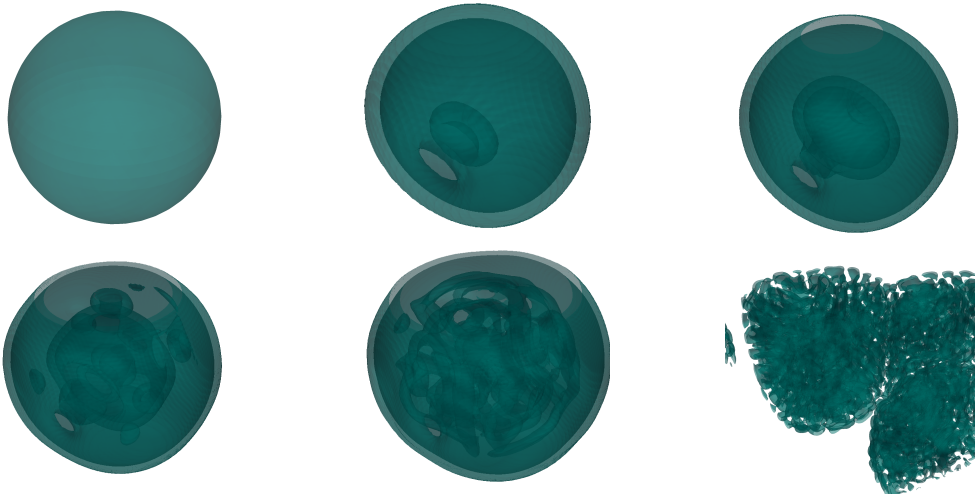
Use diagonal SDC to extend scaling

- Circumvents space-decomposition limit
- Improves strong scaling
- Enables scaling up to 4096 CPUs

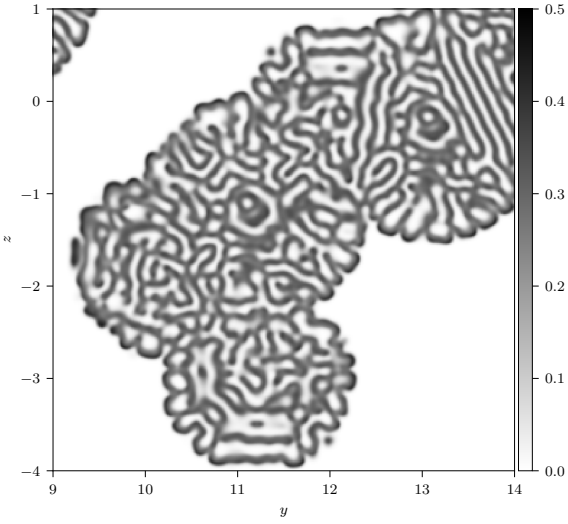
Large space-time parallel Gray-Scott simulation



Large space-time parallel Gray-Scott simulation: Zoom

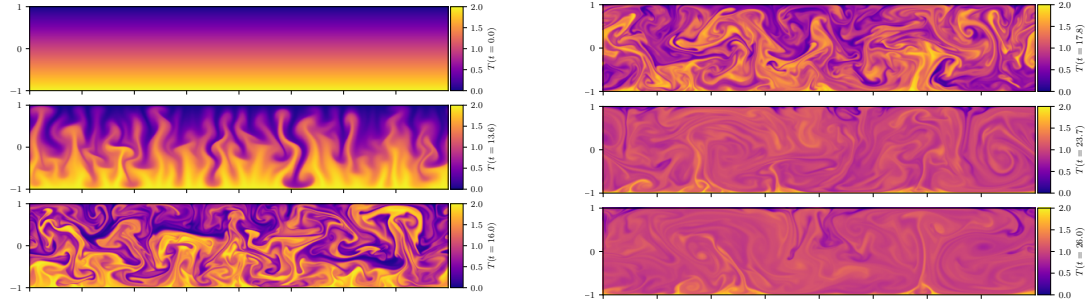


Large space-time parallel Gray-Scott simulation: 2D slice



- Turing instability: Small perturbations + diffusion = complex patterns
- Turing patterns in Gray-Scott in 3D not widely studied
- Use simulations such as this one with Δt-adaptivity and $N = 2304^3$ on $4 \times 192 = 768$ GPUs on JUWELS booster to change that

Large space-time parallel Rayleigh-Benard simulation



- Use Rayleigh number 2×10^7 on $N = 4096 \times 1024$ grid
- Use Δt-k-adaptivity with four Gauß-Radau nodes for step size selection
- Use $4 \times 1024 = 4096$ CPUs on JURECA DC (32 nodes)
- Use approx. 140k CPU hours to reach $t = 26$ because IMEX stability limit requires $\Delta t \approx 10^{-3}$

Sources I


Čaklović, G., Lunet, T., Götschel, S., and Ruprecht, D. (2024).
Improving efficiency of parallel across the method spectral deferred corrections.


Causley, M. and Seal, D. (2019).
On the convergence of spectral deferred correction methods.
Communications in Applied Mathematics and Computational Science, 14(1):33–64.


Christlieb, A., Ong, B., and Qiu, J.-M. (2009).
Comments on high-order integrators embedded within integral deferred correction methods.
Communications in Applied Mathematics and Computational Science, 4(1):27–56.


Dutt, A., Greengard, L., and Rokhlin, V. (2000).
Spectral deferred correction methods for ordinary differential equations.
BIT Numerical Mathematics, 40(2):241–266.


Glosli, J. N., Richards, D. F., Caspersen, K. J., Rudd, R. E., Gunnels, J. A., and Streitz, F. H. (2007).
Extending stability beyond CPU millennium: A micron-scale atomistic simulation of Kelvin-Helmholtz instability.
In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, New York, NY, USA. Association for Computing Machinery.

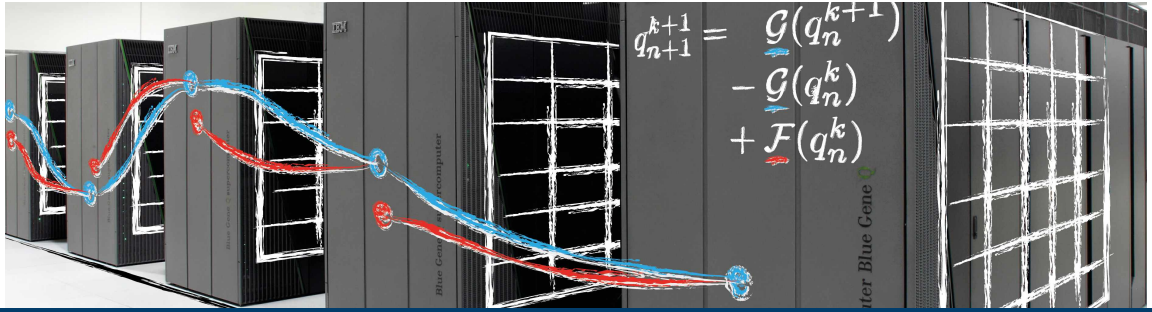

Guibert, D. and Tromeur-Dervout, D. (2007).
Parallel deferred correction method for CFD problems.
In Kwon, J., Ecer, A., Satofuka, N., Periaux, J., and Fox, P., editors, *Parallel Computational Fluid Dynamics 2006*, pages 131–138. Elsevier Science B.V., Amsterdam.

Sources II

Ruprecht, D. and Speck, R. (2016).
Spectral deferred corrections with fast-wave slow-wave splitting.
SIAM Journal on Scientific Computing, 38(4):A2535–A2557.

Schroeder, B., Pinheiro, E., and Weber, W.-D. (2011).
DRAM errors in the wild: A large-scale field study.
Commun. ACM, 54(2):100–107.

van der Houwen, P. J. and Sommeijer, B. P. (1991).
Iterated runge–kutta methods on parallel computers.
SIAM Journal on Scientific and Statistical Computing, 12(5):1000–1028.



Massively parallel adaptive spectral deferred correction in Python

December 9, 2024 | Thomas Baumann | Jülich Supercomputing Centre

Porting mpi4py-fft to GPU: Alltoall in NCCL

N,r ← Number of GPUs, MPI rank ▷ Initialize variables
sendbufs, sendbufs ← {}, {}

NCCL group start ▷ Launch all communications in a single kernel
for i in 0, 1, ..., N: **do**
    send_to ← (r + i) % N
    recv_from ← (r - i + N) % N
    sendbufs[i] ← contiguous copy of data to send
    recvbufs[i] ← contiguous empty array
    NCCL receive from recv_from
    NCCL send to send_to
end for
NCCL group end ▷ Wait for all communication to finish before unpacking

copy recvbufs to destination array

Record all of this to CUDA graph!

Porting mpi4py-fft to GPU: Alltoall in NCCL

N,r ← Number of GPUs, MPI rank ▷ Initialize variables
sendbufs, sendbufs ← {}, {}

NCCL group start ▷ Launch all communications in a single kernel
for i in 0, 1, ..., N: **do**
    send_to ← (r + i) % N
    recv_from ← (r - i + N) % N
    sendbufs[i] ← contiguous copy of data to send
    recvbufs[i] ← contiguous empty array
    NCCL receive from recv_from
    NCCL send to send_to
end for
NCCL group end ▷ Wait for all communication to finish before unpacking

copy recvbufs to destination array

Record all of this to CUDA graph!