Solvers in openCFS: Ginkgo and CUDA

openCFS User Meeting 2025

Fabian Wein[1]

Friedrich-Alexander-Universität Erlangen-Nürnberg, Department Mathematics

July 4, 2025

Solvers: Sources

- approx 10% to 50% of the total runtime is for solving the linear system(s)
- openCFS has default solvers, for large problems select manually
- some solvers can benefit from / require parameter tuning
- check the .info.xml and/or via performance.py what is going on

Textbook solvers

- standard solvers like CG, GMRES, DirectLU, etc
- direct solvers good for small problems and test suite, but not for large problems
- we have no generic geometric multigrid (GMG) solver yet

External solvers

- third party solvers like Pardiso, CHOLMOD, LIS, Ginkgo, SuperLU, etc
- highly optimized for large problems
- check license and availability

Solvers: Types

Direct solvers (Pardiso, CHOLMDO, UMFPACK, ...)

- extremely fast for small to moderate problems
- practically no tuning required
- do variants of LU factorization $\rightarrow$ additional right hand sides almost for free
- practically insensitive to ill-conditioned problems
- required memory explodes quickly for large problems (e.g. 500GB)

Iterative solvers (Ginkgo and LIS)

- Ginkgo and LIS are suites with various iterative solvers and preconditioners
- require solver/preconditioner matching the problem (default is more for robustness than speed)
- always check if the solution is converged. Tricky for ill-conditioned problems
- controllable/small memory overhead $\rightarrow$ can be used for very large problems
- in theory benefit from reordering (Metis) but Metis is slow, so better not use

Multilevel Solvers

- **multilevel solvers** are a class of iterative solvers that use multiple levels of discretization
- blazing fast ($\mathcal{O}(N)$) for large problems
- **geometric multigrid** (GMG) uses hierarchical spatial discretizations $\rightarrow$ native implementation
- **algebraic multigrid** (AMG) works as black box $\rightarrow$ less efficient
- comes with a wide range of parameters to be tuned
- algebraic multigrid is often used as preconditioner (lis, ginkgo)
- a native general purpose geometric multigrid solver would be a great project.
- probably there is a specialized multilevel solver for magnetics in openCFS?!

Pardiso

- **Par**allel **Di**rect **So**lver by Olaf Schenk (Lugano, ETH Zurich)
- easy to use and extremely fast and scalable implementation (up to moderate size)
- for definite and indefinite problems, gives even solution for non-unique problems
- part of Intel MKL (oneAPI, oneMKL), freely redistributable
- default solver in openCFS
- not available for ARM64 architecture (Apple Silicon, upcoming ARM64 Windows/Linux)

Panua-Pardiso

- newer version of Pardiso, also for ARM64
- neither open source nor free, strict license management (node locked)
- limited academic license available
- contains iterative solver, probably from M. Bollhöfer (Ilupack)
- stalled integration into openCFS

Suitesparse

- SuiteSparse mainly contains CHOLMOD and UMFPACK
- direct solvers, very fast in GPL configuration
- because of GPL license not activated by default in openCFS
- CHOLMOD $\rightarrow$ symmetric positive definite problems, UMFPACK $\rightarrow$ for general problems
- CHOLMOD can outperform Pardiso up to 20% (depending on problem)
- best approach for direct solver on ARM64 (e.g. Mac)

LIS

- **L**ibrary for **I**terative **S**olvers
- Japanese academic open source project, actively maintained
- huge collection of iterative solvers and preconditioners (including AMG)
- parallel via OpenMP, MPI is not supported in openCFS
- quite sensitive to condition and parameters w.r.t. crashes, convergence, and speed
- change settings when default configuration fails

Ginkgo

- C++17 library of iterative solvers and preconditioners
- rather new project, actively developed mainly (but not exclusively) at KIT and TUM
- BSD-3-Clause license, freely redistributable, compatible with openCFS's MIT/X11 license
- supports a wide range of solvers and preconditioners, including AMG
- OpenMP, CUDA (Nvidia), HIP (AMD), and SYCL (Intel)
- openCFS currently supports OpenMP and CUDA
- very fast, scalable and quite robust
- supports algebraic multigrid as preconditioner and solver
- option to use json file for complete generic solver configuration
- on *Please only commit to Ginkgo when GINKGO_DEVEL_TOOLS is enabled in CMake.* use `git commit -no-verify`

Test Setup

Hardware

- very fast workstation with GPU and 128 GB RAM
- Intel i9-14900KF CPU (Raptor Lake): 8 performance cores, 16 efficiency cores, 32 threads
- builds openCFS (w/o deps) in 90 sec (`make -j 20`)
- Nvidia RTX 5080 GPU with 16 GB VRAM (fp32: 56 TFLOPS, fp64: 0.9 TFLOPS (1:64))
- 50 W idle power

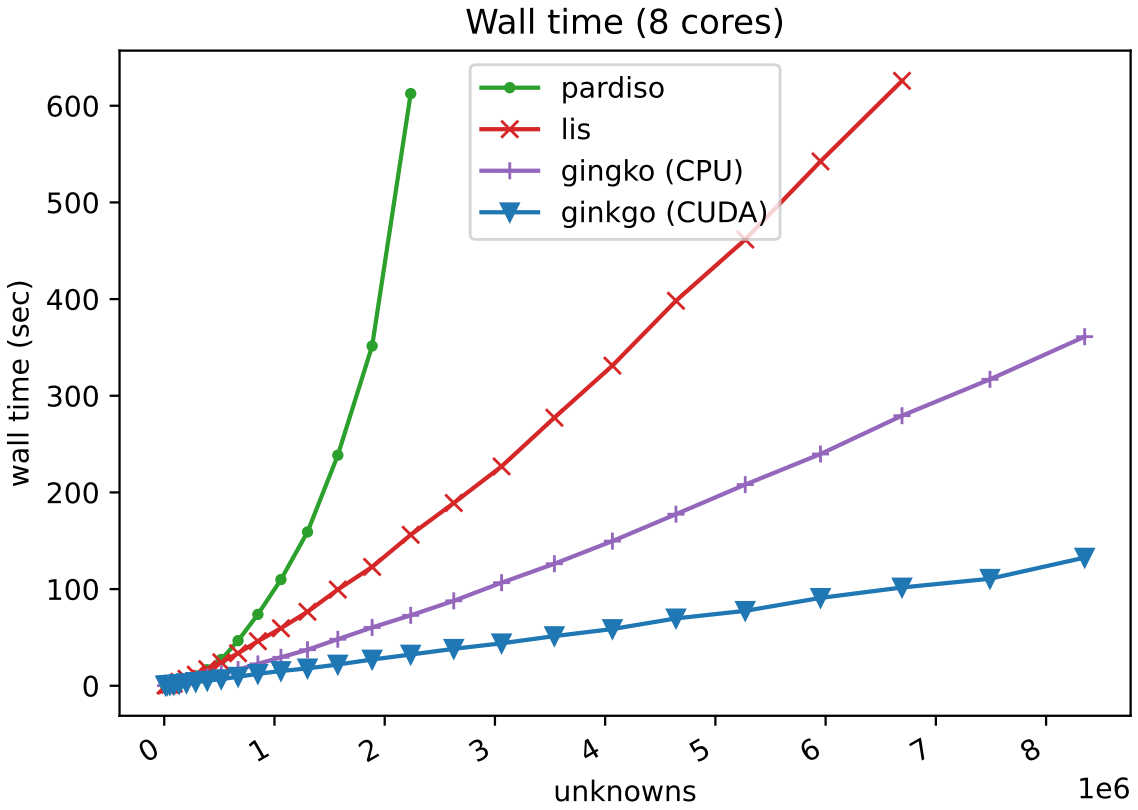
Bechmark problem

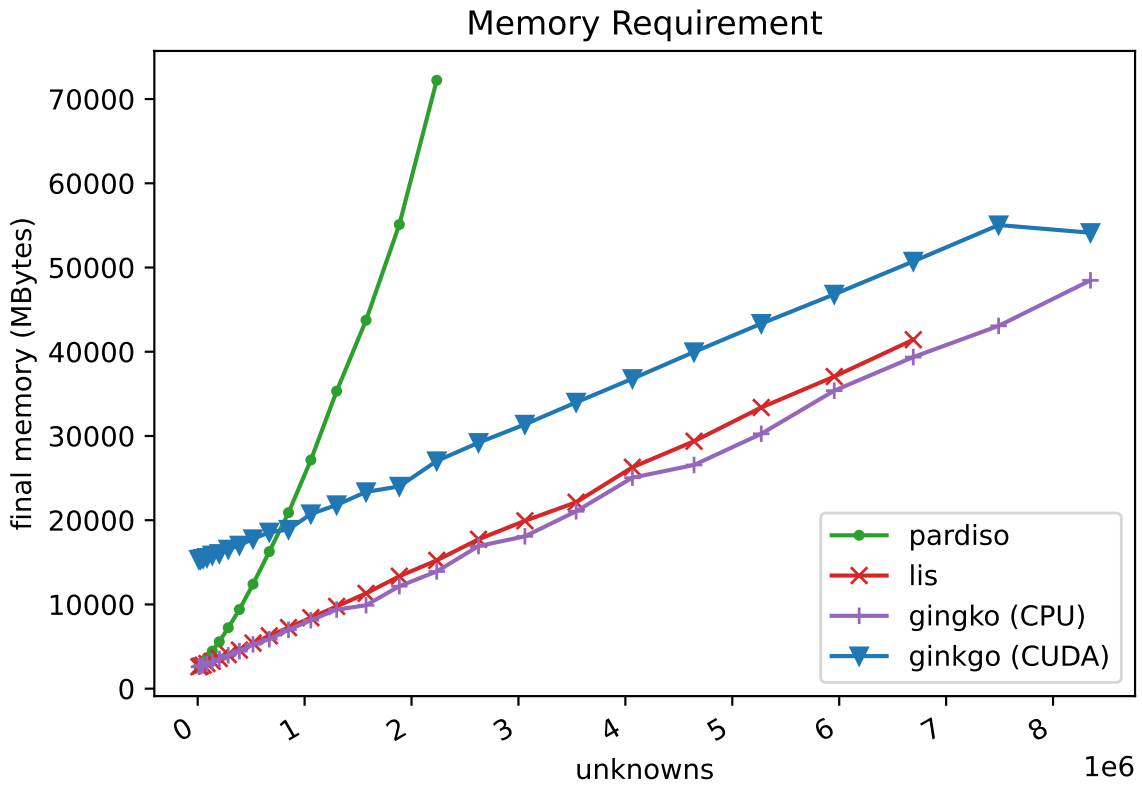
- 3D linear elasticity problem $10^3 \ldots 150^3$ elements $\rightarrow$ 3K . . . 10M unknowns
- scaled material
- we benchmark full simulation including writing results

Parameter Study

- solvers: LIS, Pardiso, CHOLMOD, Ginkgo (CPU), Ginkgo (CUDA)
- all solvers with default parameters
- 1, 4, 8, 16 threads (note we have only 8 performance cores)

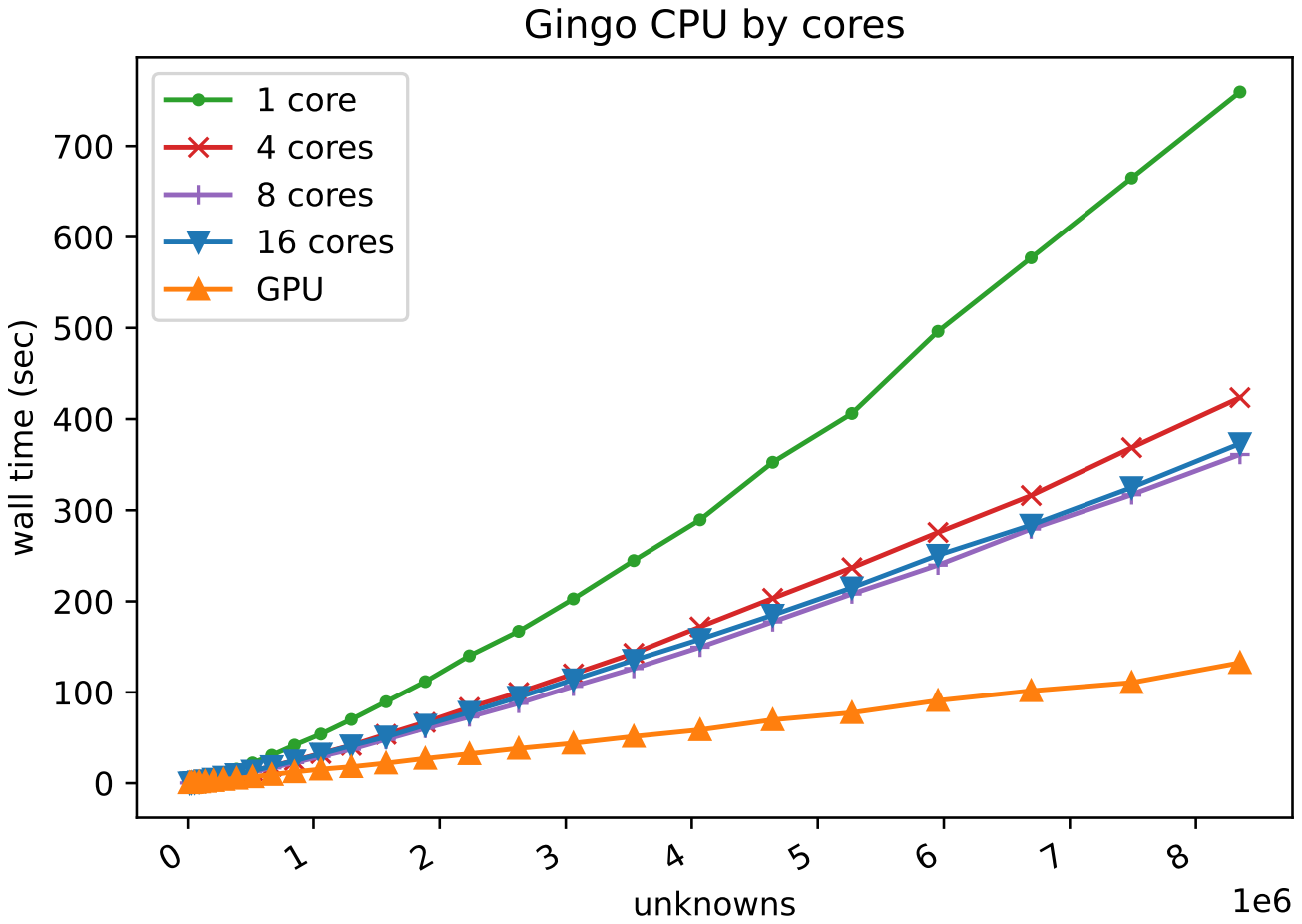
Solver Comparsion





- for large systems direct solvers need too much memory and are slower
- time includes openCFS overhead, Ginkgo (CUDA) is about 10% of total wall time

Ginkgo (CPU) by Cores



- Ginkgo with OpenMP does not scale well with too many cores

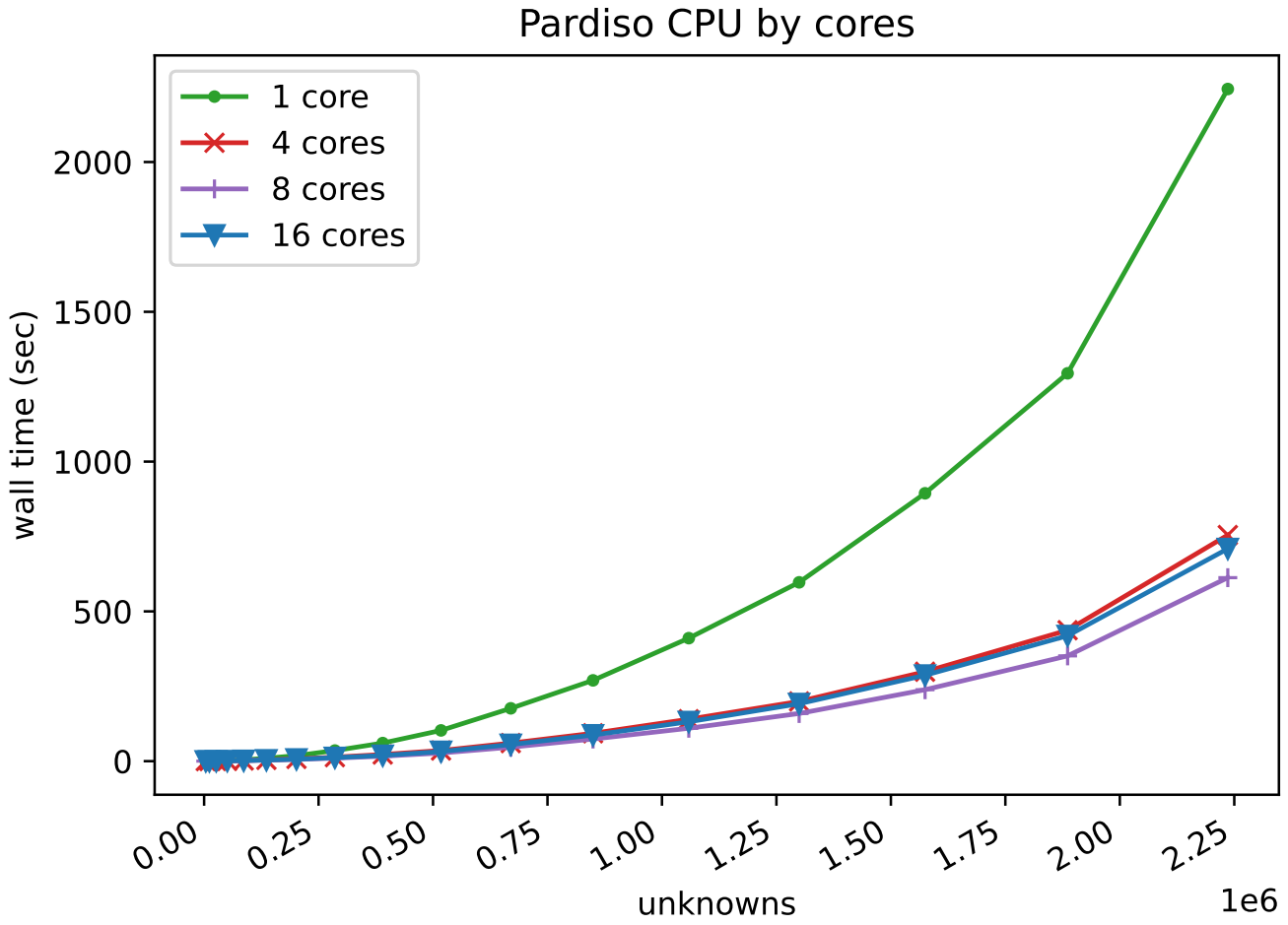
Amdahl's Law

- wall time in sec for 90^3 problem (2.2M unknowns) with 8 cores

	Pardiso	Ginkgo (OpenMP)	Ginkgo (CUDA)
total	613 (100%)	73 (100%)	32 (100%)
graph_setup	14 (2.3%)	13 (17%)	14 (43%)
init_static	20 (3.3%)	19 (26%)	20 (63%)
solver setup	580 (95%)	0.04 (0%)	1.1 (3.3%)
solver solve	5 (0.8%)	46 (63%)	2.9 (9%)

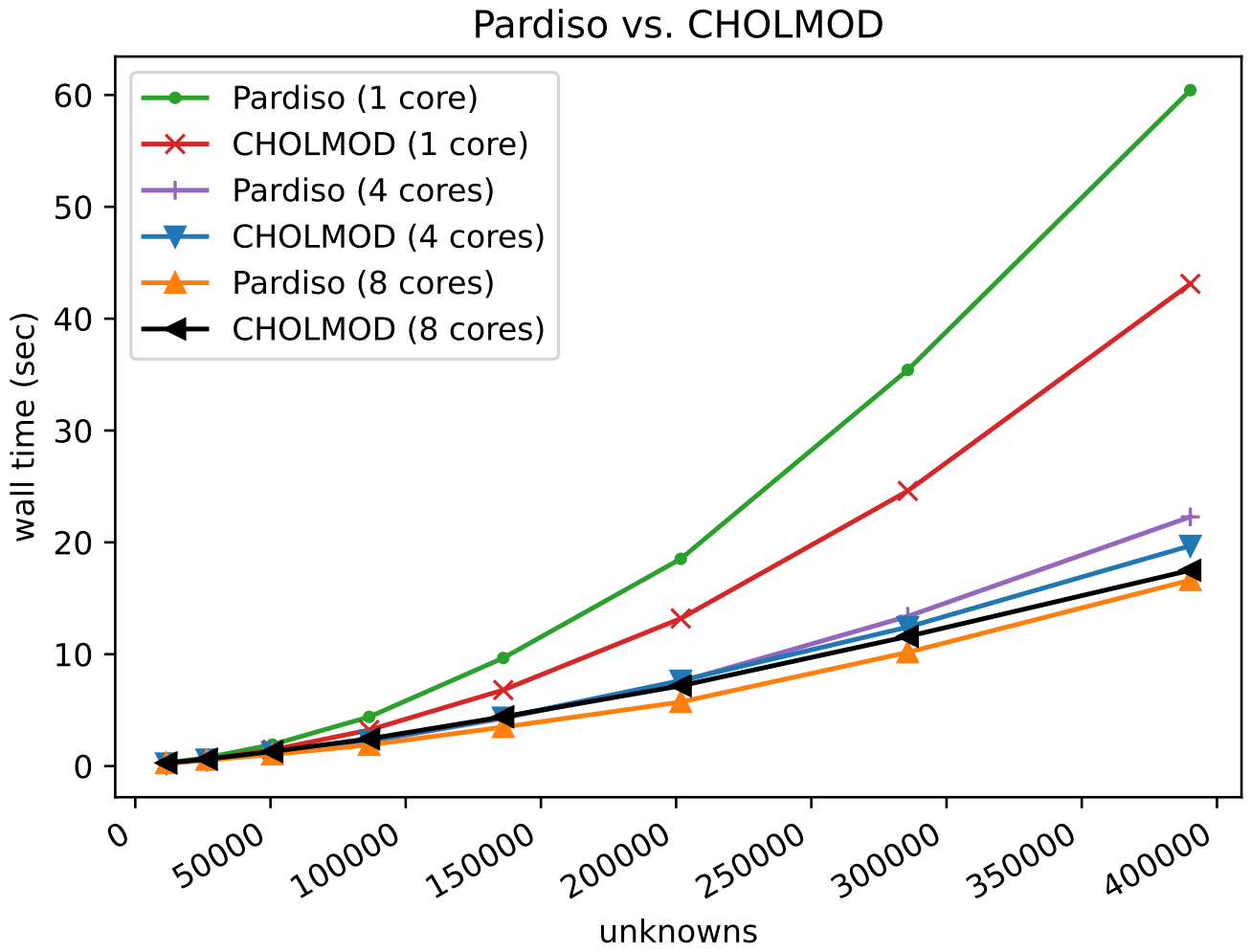
- *when we efficiently parallelize parts of a program, the speedup is limited by the serial part*
- Ginkgo (CUDA) is 200 times faster than Pardiso, and 15 times faster than Ginkgo (OpenMP)
- it is worth to tune Ginkgo (OpenMP) for speed
- huge speedup for several solved systems (transient problems / optimization)
- reordering by Metis is 37 sec w/o changing Ginkgo performance
- use `performance.py problem.info.xml` to see timing statistics

Pardiso CPU by Cores



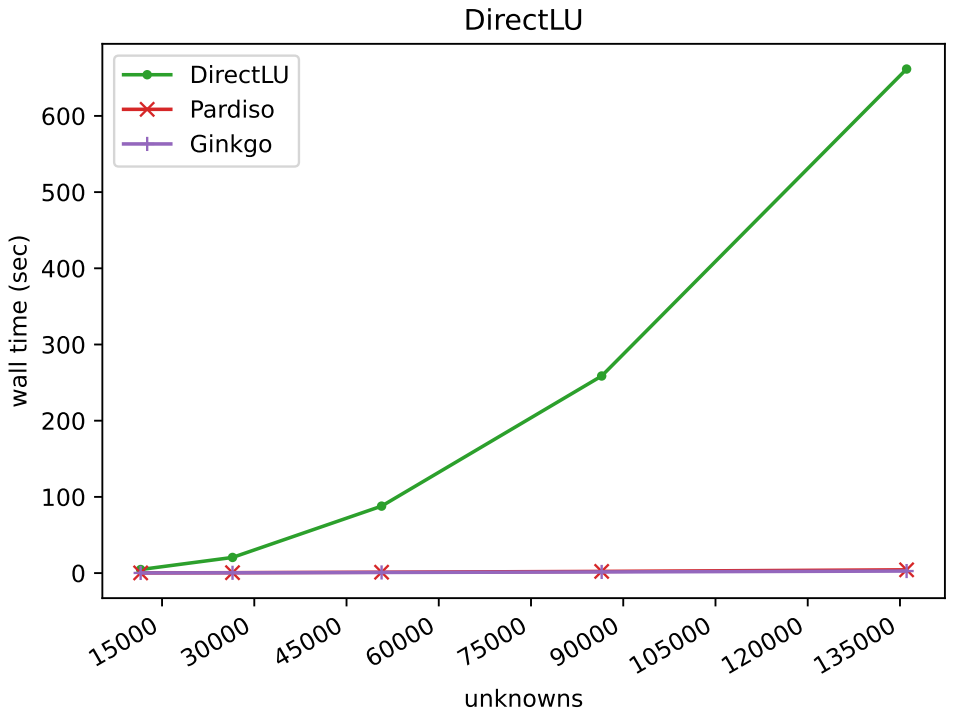
- Pardiso quickly saturates on the test system

Pardiso vs. CHOLMOD



- for few cores CHOLMOD is significantly faster than Pardiso (for this problem)

DirectLU



- openCFS's DirectLU text book solver is sometimes set as solver in the test suite
- it makes sense to have a defined, always available solver in the test suite (even fastest!)
- it makes no sense at all to use DirectLU (or any other text book solver) for non-toy problems

Using Ginkgo

usage of Ginkgo with default settings

```
  ...
  </pdeList>
  <linearSystems>
    <system>
      <solverList>
        <ginkgo/>
      </solverList>
    </system>
  </linearSystems>
</sequenceStep>
```

translates to

```
<ginkgo precond="jacobi" solver="cg" cuda="auto" initial_zero="false" maxIter="10000" minimalTolerance="1e-8"
   tolerance="1e-9" toleranceType="absolute" fp32="false" dump_profiling="false">
  <jacobi max_block_size="1" />
</ginkgo
```

Using Ginkgo cont.

check .info.xml

```
<ginko exec="openmp" cuda_available="no">
...
<precond type="jacobi" max_block_size="1"/>
<solver type="cg" max_iter="10000" tolerance="1e-09" mode="absolute" precision="64"/>
<solve analysis_id="iter:0" converged="yes" iters="188" residual="8.741e-10" sol="9708.1" initial="0"/>
```

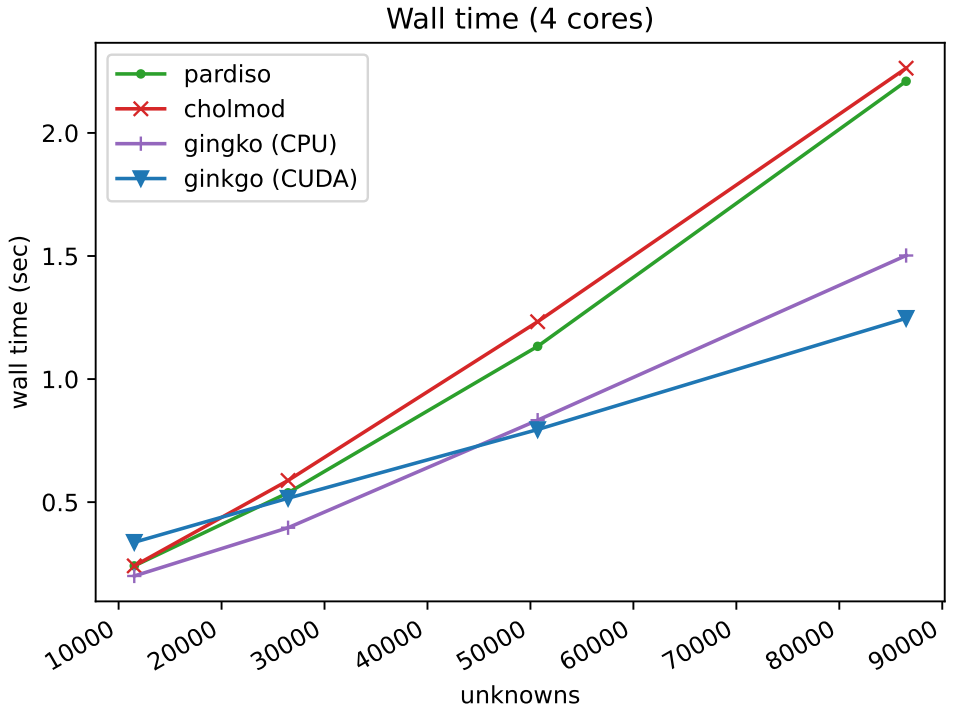
- preconditioners: **jacobi**, amg, ic, ilu, json
- solvers: **cg**, gmres, bicgstab, direct
- cuda: **auto**, off, on (compile with USE_CUDA=ON only when locally available)
- tolerance: 1e-9 (target)
- minimalTolerance: 1e-8 (acceptable)
- initial_zero: **false**, true (use last solution as initial guess or start from 0)
- fp32: **false**, true (use single precision - almost no benefit and slightly more iterations)

Advanced Ginkgo

- Ginkgo documentation has (large) room for improvement $\rightarrow$ skip doxygen, use source files
- select e.g. gmres for non positive definite problems
- AMG as preconditioner: doubled performance for 2D elasticity, stalled for 3D
- use json configuation for advanced configuration, e.g. `<ginkgo json="cg.json" />`
- in Testsuite/TESTSUITE/Solver/Ginko/json (taken from Ginko's source examples)
  - cg.json
  - example_blockjacobi-cg.json
  - example_mixed-pgm-multigrid-cg.json
- HIP (AMD's variant of CUDA) would be trivial to add

Take Aways

- for Ginkgo with CUDA use 1 thread. Performance is same, but memory usage slightly lower
- Ginkgo seems to have no GPU memory limitation (automatic memory offloading?)
- Ginkgo is much more robust compared to LIS (but most important is configuration)
- it is worth using Ginkgo even for moderate problems



The End

Any Questions?