Date: December 17, 2018

To: Lois Curfman McInnes

cc: Xiaoye Sherry Li, Yang Liu (LBNL); Tzanio Kolev (LLNL); Mihai Anitescu (ANL)

From: Pieter Ghysels, 2.3.3.07/STRUMPACK-SuperLU

Milestone Deliverable — ST-MS-08-1100

Milestone Due Date: October 31, 2018

Milestone Completion Date: October 31, 2018

Description of Milestone:

After general improvements of both solvers in Year 1, we will investigate the specific weak areas of the solvers for the ECP applications. We expect further improvement of the codes with application-specific tuning.

We have selected two ECP applications for this bottleneck study:

- CEED Co-Design Center: Indefinite Maxwell simulation using MFEM, see Section 1.1.
- ExaSGD: Optimizing Stochastic Grid Dynamics, see Section 1.2.

Completion Proof of the Milestone:

We wrote a report detailing the both STRUMPACK and SuperLU performance for the typical problems from the two ECP applications. See the rest of this memo for technical details.

Tasks to Complete the Milestone:

Write a report with clear identification of future development focuses in both solvers.

Execution Plan:

- Select 2-3 ECP applications, interview the teams to find out the numerical properties of their linear systems and the accuracy and performance needs from the solvers.
- Obtain the matrices from the application teams and perform standalone testing of SuperLU and STRUMPACK.

- Identify performance bottlenecks in the solvers components, propose mitigation strategies.
- Work with the application teams to build realistic performance models to estimate the overall performance of the applications, and perform model-based code optimization. We will develop the Roofline models for on-node analysis and alpha/beta-models for inter-node communication analysis.

Milestone Dependencies: For a previous shared milestone with project AD-1.2.5.3.5-CEED(MR), we developed an interface from MFEM (mfem.org), to STRUMPACK. This interface closely mimics the interface from MFEM to SuperLU-Dist, which was already available.

Person(s) Responsible for Completing the Milestone:

Pieter Ghysels, Xiaoye Sherry Li, Yang Liu (LBNL)

Thanks to Mark Stowell and Tzanio Kolev (LLNL) from the MFEM/CEED project for providing the MFEM driver application `ex3p_indef`, an indefinite Maxwell mini-app.

1 Technical Work Scope, Approach, Results

In Section 1.1 we present results for SuperLU and STRUMPACK within an indefinite Maxwell simulation with MFEM. In Section 1.2, we look at the ExaSGD application.

1.1 MFEM – Indefinite Maxwell

The MFEM (mfem.org) team, specifically Tzanio Kolev and Mark Stowell (LLNL), provided us with an example code: `ex3p_indef.cpp`. For this milestone, we run experiments with the `ex3p_indef` example code, evaluating performance of SuperLU and STRUMPACK, through the interfaces from MFEM directly to both solvers. The original example code `ex3p.cpp`, which is part of the main MFEM code distribution, solves a simple electromagnetic diffusion problem corresponding to the second order definite Maxwell equation

$$\nabla \times \nabla \times E + E = f \tag{1}$$

with some given tangential field as boundary condition for E. For this definite Maxwell test problem, the MFEM team has good preconditioners available. For instance the Auxiliary space Maxwell Solver (AMS) works well as a preconditioner for CG/GMRES.
The `ex3p` example was modified to `ex3p_indef` by Tzanio Kolev and Mark Stowell, to make it indefinite. For the indefinite case, no good preconditioners are available, and the problem is typically solved with a direct solver, such as SuperLU. The strong form of the problem being solved in `ex3p_indef` is

$$\nabla \times \frac{1}{\mu} \nabla \times E + \sigma E = f \tag{2}$$

and the corresponding weak formulation is

$$\left(\frac{1}{\mu}\nabla\times E,\nabla\times E'\right)+\left(\sigma E,E'\right)=\left(f,E'\right)\,, \tag{3}$$

with $\mu = 1$. The right hand side is computed as $f = (\kappa^2 - \Omega^2)\sin(\kappa x)$ with $\kappa = \frac{\Omega}{2}$. With $\sigma = 1$, this corresponds to the easy second order definite Maxwell problem. With $\sigma = -\Omega^2$ this is the indefinite Maxwell problem, in the frequency domain. The problem becomes harder with increasing Ω. This example can be run with different input geometries and different types of meshes: triangular, quadrilateral, tetrahedral, hexahedral, surface and volume meshes. It always uses Nedelec elements of order 1. The required minimum number of serial refinements is set in the code, depending on Ω as

$$\text{serial_refinements} = \lceil \log_2 \frac{10\Omega}{2\pi} \rceil\,, \tag{4}$$

in order to ensure a minimum number of degrees of freedom per wavelength, and additionally, 2 levels of parallel refinement are always performed.

The example code `ex3p_indef` is not a fully fledged mini-app, but more like a basic code example demonstrating the use of H(curl) finite element spaces with the curl-curl and the (vector finite element) mass bilinear form. The example code performs the following steps:

- Read mesh description file, construct a serial mesh and refine the serial mesh for a given number of uniform refinements (dependent on the value of Ω)
- Construct a parallel mesh from the serial mesh and further refine the parallel mesh
- Construct a finite element space on the parallel mesh
- Determine the list of true essential boundary dof's
- Setup the parallel linear form for the right-hand side
- Define the solution vector x as a parallel finite element grid function
- Set up the parallel bilinear form corresponding to the EM diffusion operator
- Assemble the parallel bilinear form and the corresponding linear system
- Solve the linear system
- Save and visualize the solution

For this milestone we focus on the solution of the linear system, as this takes most of the time.

In all tests with either SuperLU or STRUMPACK, we disable static pre-pivoting using the MC64 column permutation. The MC64 code finds a column/row permutation that maximizes the diagonal elements of the sparse matrix. Maximizing the diagonal elements improves numerical stability of the solvers, and it might be required when no (row) pivoting is performed during the sparse

Gaussian elimination. However, for the problems considered here, it is not required. The MC64 static pivoting can be disabled from the MFEM interfaces to either SuperLU:

```
superlu->SetRowPermutation(superlu::NOROWPERM);
```

or STRUMPACK:

```
strumpack->SetMC64Job(strumpack::MatchingJob::NONE);
```

1.1.1 SuperLU and STRUMPACK Direct Solvers

In Figures 1 and 2, we present the times to solve a linear system with either SuperLU-Dist 5.1.3, SuperLU-Dist master (commit 275d212032c91528491109f413ffa5ba6ef071e1, Oct 23, 2018), and with the latest release of STRUMPACK (3.1.1). The problem being solved is the indefinite Maxwell setup described above with $\Omega = 5$. Figures 1 and 2 both show time spent in the numerical factorization phase and the triangular solver phase. Since both SuperLU and STRUMPACK are setup as direct solvers, there is a single triangular solve phase (solving with both L and U factors). For this setup, GMRES with the AMS preconditioner did not converge within 500 iterations, the GMRES residual had stagnated.

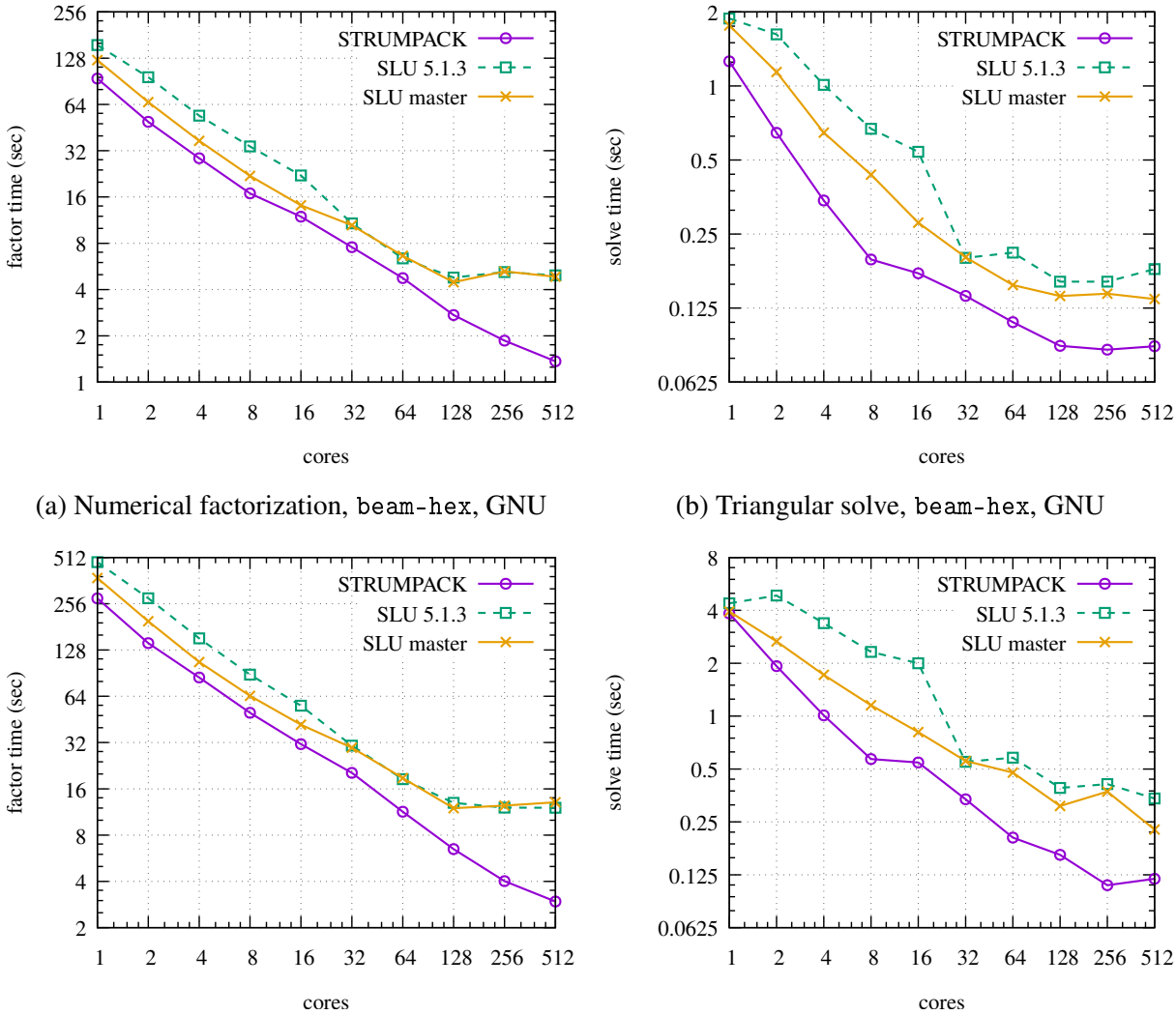
Figure 1 shows results on NERSC's Cori/Haswell, using the GCC 7.3.0 compiler and using cray-libsci/18.03.1 for BLAS/LAPACK/ScaLAPACK. Figure 2 shows results on NERSC's Cori/Haswell, using the Intel 18.0.1 compiler and MKL 2018.1.163 for BLAS/LAPACK/ScaLAPACK. Especially for the GNU compiler we see significant improvement with the latest SuperLU version, compared to the older 5.1.3 release, which is most pronounced within a single node (1-32 cores), see Figure 1 (b) and (d). We observe slightly better scaling for STRUMPACK than for SuperLU for the numerical factorization, with the SuperLU scaling leveling of after about 4 nodes. For the triangular solve, especially with the Intel compiler, see Figure 2 (b) and (d) scaling is better for SuperLU than for STRUMPACK. This is due to several recent optimizations in SuperLU's triangular solve, which are included in the 6.0.0 release. These optimizations include the use of broadcast and reduction communication trees; for more details we refer to the ST-MS-08-1120 milestone and the accompanying publication[1].

Figure 3 shows the attained floating point throughput for the same experiments as shown in Figures 1 and 2. On a single core, the numerical factorization comes very close to the peak floating point performance. On 512 cores, 16 nodes, this strong scaling experiment still achieves about 16 % of peak. For the triangular solve phase, we typically monitor performance in terms of maximal attained bandwidth.

1.1.2 SuperLU Supernode Relaxation and Parallel Symbolic Factorization

By default, the interface from MFEM to SuperLU-Dist uses the parallel fill-reducing ordering ParMETIS (the parallel version of METIS) inside SuperLU-Dist. This parallel ordering in SuperLU-

[1] Yang Liu et al. "Highly scalable distributed-memory sparse triangular solution algorithms." 2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing. Society for Industrial and Applied Mathematics, 2018.



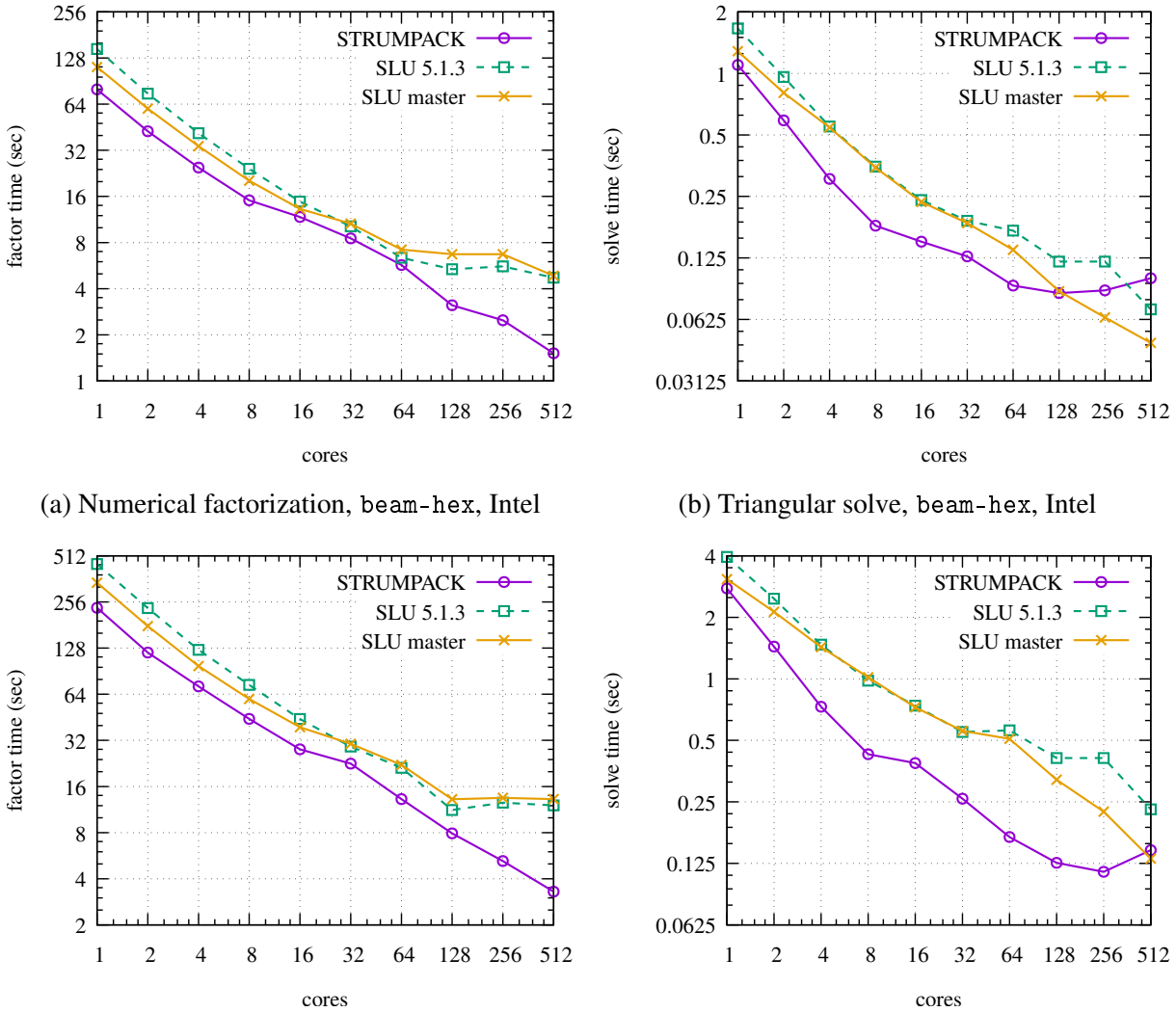
(a) Numerical factorization, `beam-hex`, GNU

(b) Triangular solve, `beam-hex`, GNU

(c) Numerical factorization, `beam-tet`, GNU

(d) Triangular solve, `beam-tet`, GNU

Figure 1: Strong scaling experiment on `beam-hex.mesh`: $821,568$ finite element unknowns, $26,196,288$ nonzeros in the system matrix and on `beam-tet.mesh`: $1,887,552$ finite element unknowns, $30,408,000$ nonzeros in the system matrix. $\Omega = 5$, `METIS` fill-reducing reordering. GCC 7.3.0, and using cray-libsci/18.03.1.



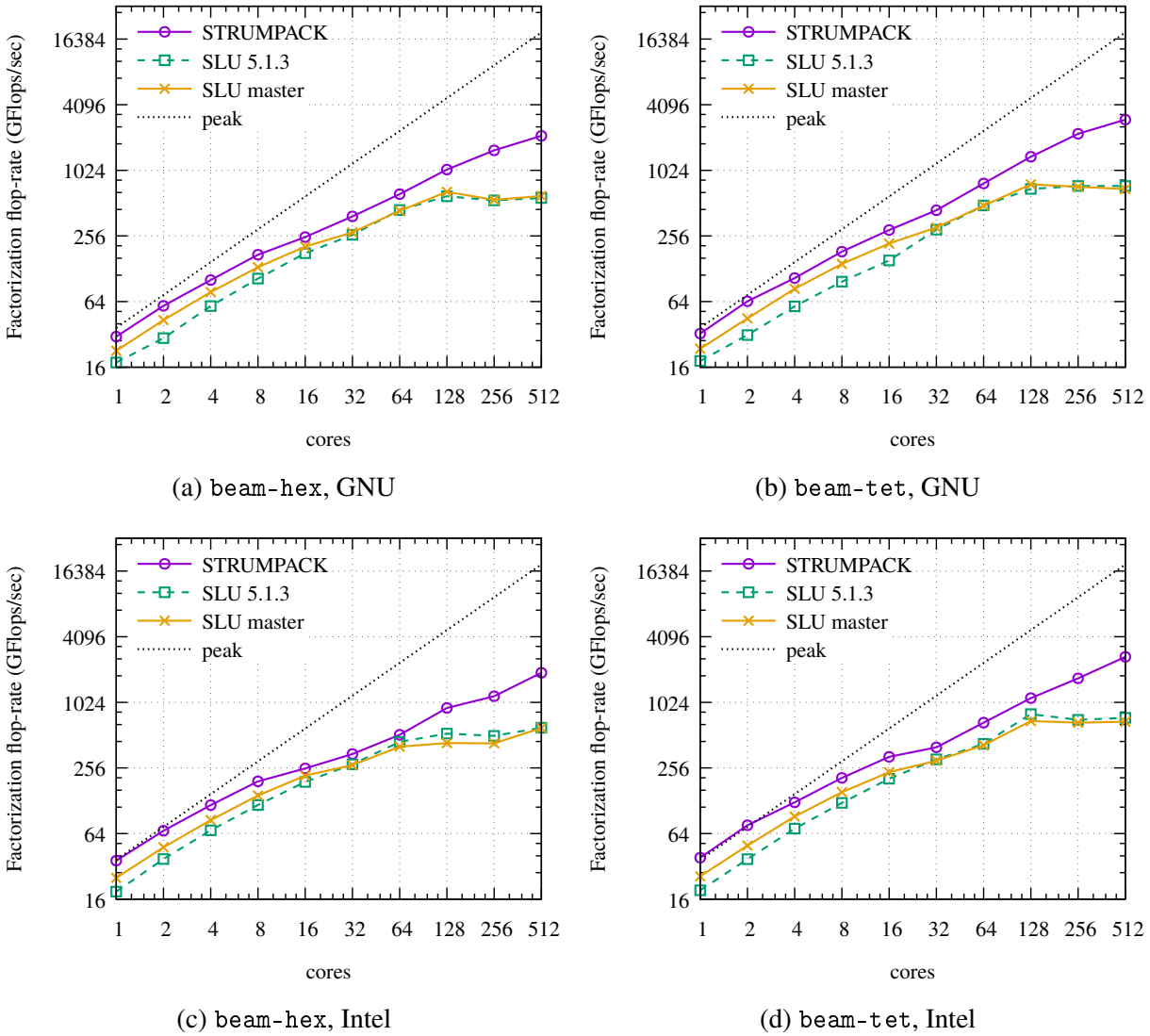
(a) Numerical factorization, `beam-hex`, Intel

(b) Triangular solve, `beam-hex`, Intel

(c) Numerical factorization, `beam-tet`, Intel

(d) Triangular solve, `beam-tet`, Intel

Figure 2: Strong scaling experiment on `beam-hex.mesh`: $821,568$ finite element unknowns, $26,196,288$ nonzeros in the system matrix and on `beam-tet.mesh`: $1,887,552$ finite element unknowns, $30,408,000$ nonzeros in the system matrix. $\Omega = 5$, `METIS` fill-reducing reordering. icpc (ICC) 18.0.1 and MKL 2018.1.163.



(a) `beam-hex`, GNU

(b) `beam-tet`, GNU

(c) `beam-hex`, Intel

(d) `beam-tet`, Intel

Figure 3: Flop-rate in the numerical factorization for a strong scaling experiment: at 512 cores, there are only about 1604 (a) or 3686 (b) DoF's per core. Theoretical peak on 512 cores, 16 nodes, is 18841.6 GFlop/s, or 36.8 Gflops/core.

Dist is combined with a parallel symbolic analysis phase which proceeds the actual numerical factorization (the sparse Gaussian elimination).
SuperLU constructs an elimination tree from the permuted sparse matrix. This elimination tree prescribes the dependencies between rows in the matrix, and guides the order of the (parallel) elimination of rows. Nodes in the tree correspond to columns in the matrix with the same sparsity pattern can be grouped together. This reduces the size of the elimination tree, now called the assembly or supernodal tree. Nodes in the supernodal tree correspond to multiple columns in the sparse matrix, which allows the use of higher level BLAS routines, and greatly improves efficiency. SuperLU also implements so-called supernode relaxation, in which nodes for columns with a similar but not exactly the same sparsity pattern can still be combined into supernodes. This allows more high level BLAS calls, but leads to more fill, i.e., more nonzeros in the sparse triangular factors, and higher memory usage. This relaxation can be controlled with the `NREL` environment variable, which is set by default to 20:

```
export NREL=100
```

A value of 100 means that at the leafs of the supernodal tree, nodes which are smaller than 100 can be grouped together. This means that a larger value of NREL will lead to less supernodes, and a smaller elimination tree (the graph/DAG G(L) or G(U)). However, it also leads to more fill, i.e., more nonzeros in L and in U, and a higher peak memory usage. This can be seem in Table 1, where we look at the influence of the `NREL` parameter when using either the METIS or ParMETIS fill-reducing reorderings, for two different values of the `NREL` parameter. However, the NREL parameter does not seem to have any effect when using ParMETIS. The fact that the NREL parameter is not used in SuperLU-Dist when running the parallel symbolic factorization (enabled with ParMETIS reordering) leads to much lower efficiency during the numerical factorization.
STRUMPACK does not perform supernode relaxation, so it has a relatively large amount of supernodes. However, since STRUMPACK is a multifrontal algorithm, while SuperLU is a purely Supernodal implementation, it's communication pattern is different.
In the SuperLU interface, one can disable parallel symbolic factorization:

```
// options->ParSymbFact = YES;
options->ParSymbFact = NO;
```

to allow supernode relaxation. However, as can be seen in Figure 4, in this case the sequential symbolic factorization becomes a bottleneck.
In a future SuperLU release, we will add supernode relaxation in the parallel symbolic factorization.

1.1.3 Larger Scale Experiment – Graph Algorithm Bottlenecks

In Figure 5 we illustrate parallel scalability of a linear solve with STRUMPACK as a direct solver on up to 8129 cores. Numerical factorization (which has received much attention) scales well. The solve phase does not scale in this strong scaling regime, but the overall time spent in solve is very small (especially relative to the factorization). Time for the symbolic analysis phase starts to increase. We plan to make improvements to the symbolic analysis phase: aggregate some messages and exploit asychronous communication. The fill-reducing ordering codes do not scale: ParMETIS

	SuperLU 6.0.0 NREL=20		SuperLU 6.0.0 NREL=100		STRUMPACK	
	METIS	ParMETIS	METIS	ParMETIS	METIS	ParMETIS
Nonzeros in L	48,118,782	35,754,812	106,755,888	35,754,812		
Nonzeros in U	48,118,782	34,179,640	106,755,888	34,179,640		
nonzeros in L+U-I	-	69,934,452	-	69,934,452		
nonzeros in L+U	94,662,396	-	211,936,608	-	75,495,336	74,873,648
nonzeros in LSUB	18,654,322	-	13,691,743	-		
No of supers	230,495	1,051,567	39,005	1,051,567	857,273	858,829
Size of G(L)	5,202,967	9,553,059	2,994,746	9,553,059		
Size of G(U)	3,627,799	8,501,492	1,419,578	8,501,492		
Size of G(L+U)	-	18,054,551	-	18,054,551		
Total highmark (MB)						
Sum:	17876.87	15776.47	17620.80	15776.47		
Avg:	558.65	493.01	550.65	493.01		
Max:	574.41	499.23	629.86	499.23		
Factor Mflops	3726.40	865.81	17124.29	864.39	29882 .7	28890.1
Factor time (s)	3.35	12.90	1.37	12.92	0.397	0.397

Table 1: Total amount of nonzeros in the triangular factors (the fill) and statistics of the assembly tree for the indefinite Maxwell problem on `beam-tri.mesh`, using 32 MPI ranks on a single node of Cori Haswell, with (Par)METIS version 4.0.3. The input sparse matrix has 1,575,168 rows and columns and 7,866,624 nonzeros. When using SuperLU with METIS, increasing NREL leads to less (but bigger) supernodes, better performance, but more fill and higher memory usage. The experiments with ParMETIS use SuperLU's parallel symbolic factorization. Currently, the NREL parameter has no impact in SuperLU when using parallel symbolic factorization. Unfortunately STRUMPACK does not (yet) keep track of peak memory usage.

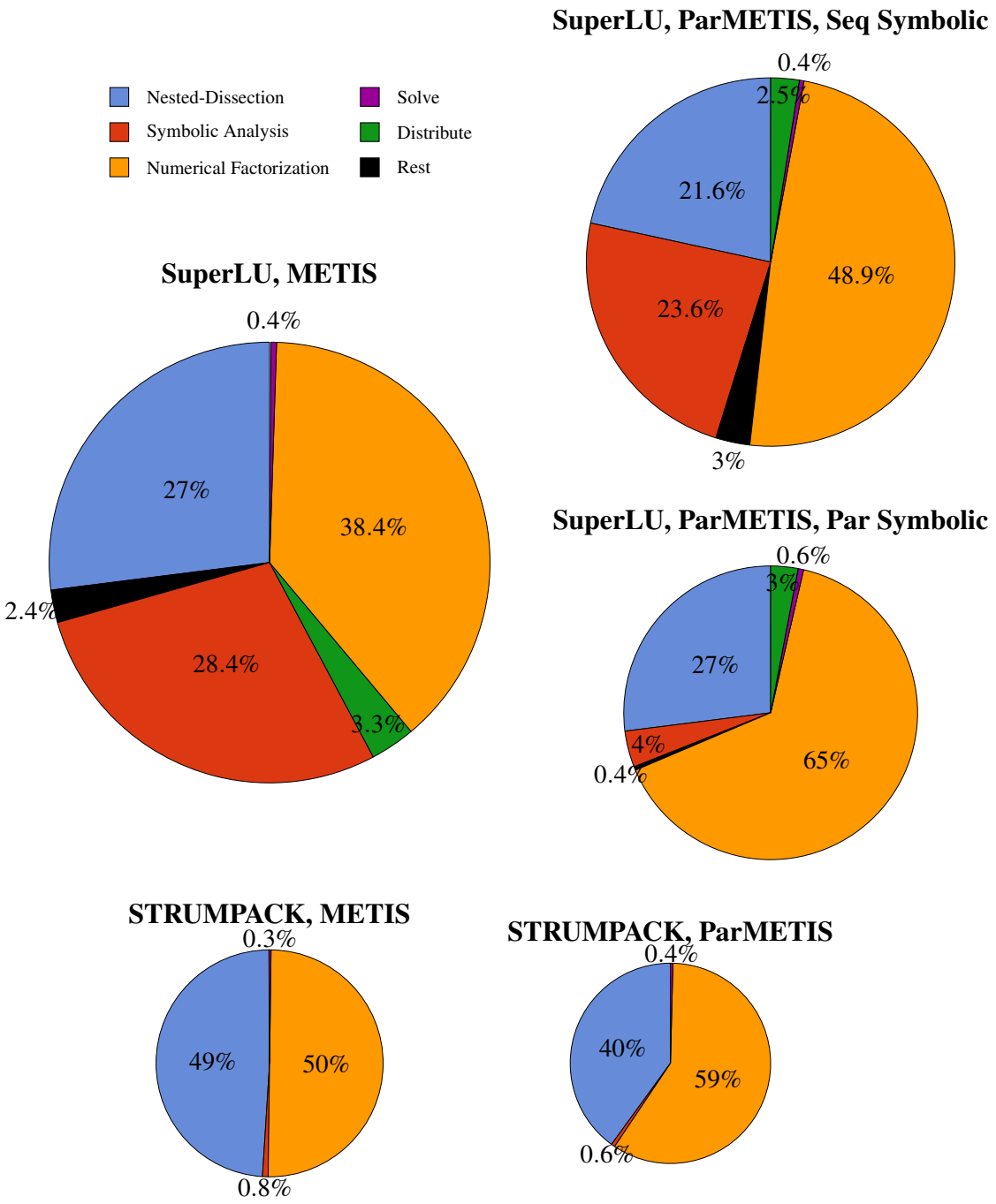


Figure 4: For mesh `periodic-cube.mesh`, on 512 cores on Cori/Haswell, 16 nodes, using GCC 7.3.0 and cray-libsci/18.03.1. Radius of each disc is proportional to total time. As can be seen from the STRUMPACK results, the speedup from METIS to ParMETIS (blue), is minimal. For SuperLU, we ran with METIS (blue) with sequential symbolic factorization (red), and with ParMETIS (blue) in combination with both sequential and parallel symbolic factorization (red). Sequential symbolic factorization can become a bottleneck.

becomes a serious bottleneck. The sequential METIS code is faster than ParMETIS at large scale. Moreover, numerical factorization is faster when using METIS than when using ParMETIS, because METIS generates orderings of better quality. One possible improvement would be to call ParMETIS on a subcommunicator, for instance limited to a handfull of nodes. We will implement this strategy in a future STRUMPACK release, with the number of ranks to use for ParMETIS a user tunable parameter. More nodes might be required due to memory restrictions.
As an alternative, we are also developing a new ordering code, based on spectral nested-dissection. We have some encouraging results with this new (shared memory) code[2]. By leveraging (scalable) eigensolvers, we hope this code will also scale better (than the current state-of-the-art) to multiple distributed memory nodes. This work is currently funded by FASTMath. The code will be released under a BSD license. This will also remedy licensing issues that some users (in industry) currently face with ParMETIS.

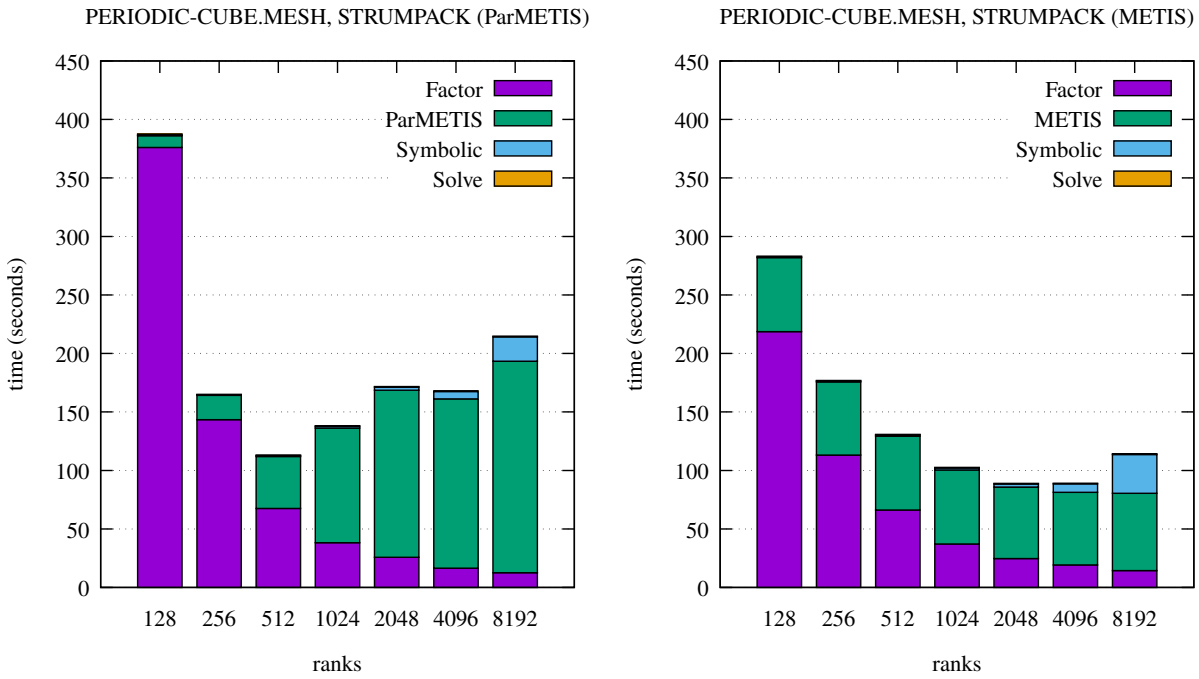


Figure 5: At large scale, ParMETIS becomes a serious bottleneck. Numerical factorization scales well. Numerical factorization is faster when using sequential METIS, due to the fact that the ordering produced by METIS is better than the ParMETIS ordering. The symbolic analysis phase has scalability issues at large scale.

1.1.4 STRUMPACK Rank-Structured Preconditioner

Figure 6 illustrates the STRUMPACK preconditioning strategy. It shows the sparsity pattern of the filled matrix, i.e., the sparsity pattern of $L+U$, where $AP = LU$ with A the original sparse input matrix, P a permutation matrix, L a lower triangular matrix and U an upper triangular matrix.

[2]P. Ghysels et al., "Scalable Krylov Methods for Spectral Graph Partitioning", 18th SIAM Conference on Parallel Processing for Scientific Computing. `http://meetings.siam.org/sess/dsp_talk.cfm?p=89316`

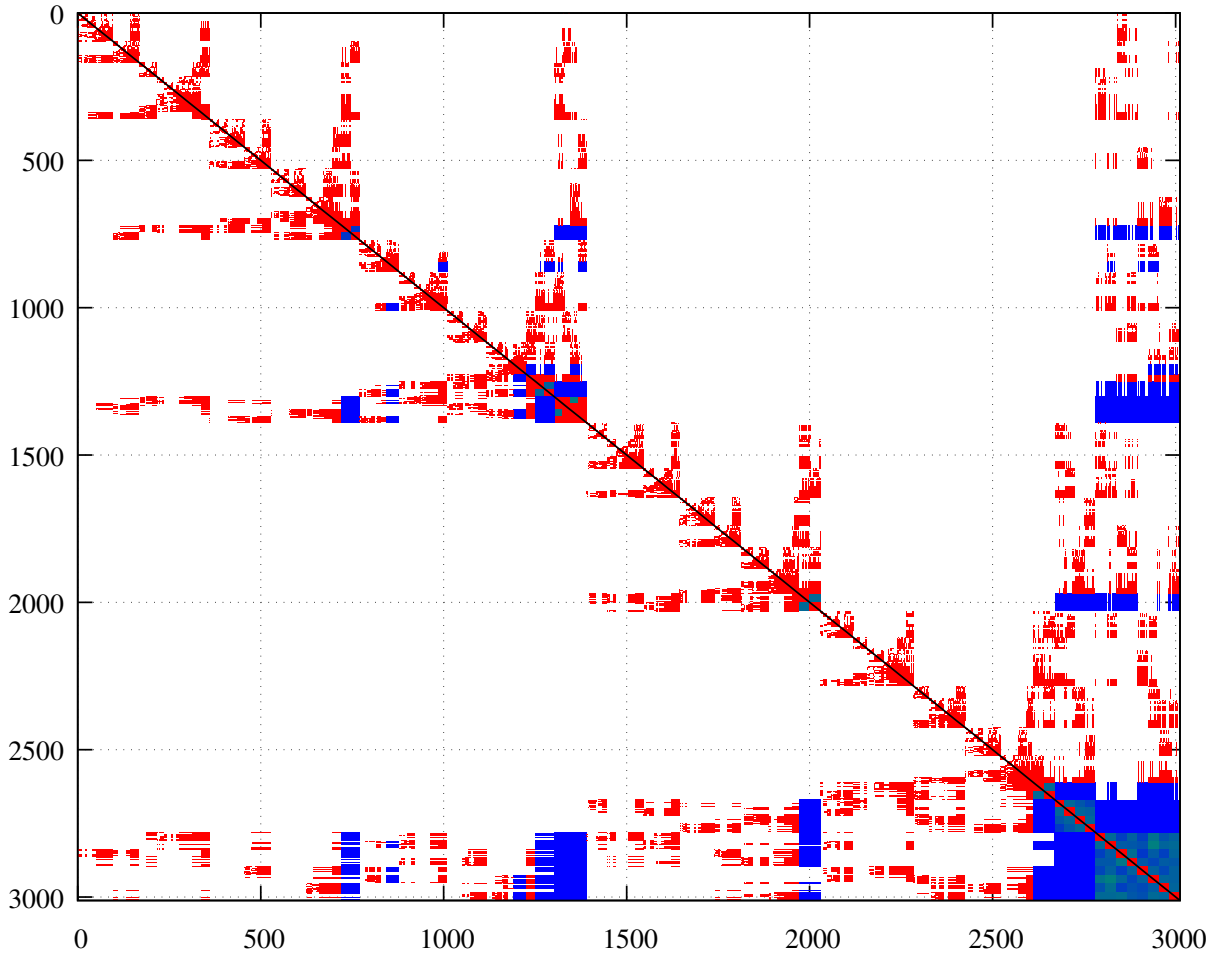


Figure 6: Illustration of the sparse matrix after factorization, i.e., the sparsity pattern of L+U, with BLR compression. Blue blocks denote low-rank tiles in the BLR compressed format, with the color going to red for increasing ranks. This is for `amr-hex.mesh` with BLR leaf size 16 and minimum separator size for BLR compression 32. In practice we take larger values for leaf size and minimum separator size, since compression does not pay off for such small matrices.

The matrix is reordered using a fill-reducing ordering, for instance nested-dissection. The larger dense sub-blocks in L and U (colored in blue in Figure 6) are the fill-in generated during the Gaussian elimination process. The rank-structured preconditioner in STRUMPACK approximates these dense blocks with so-called rank-structured matrices: the matrices are split into smaller blocks and certain 'admissible' blocks are compressed using low-rank approximation. This reduces memory usage as well as floating point operations. Typically, we have used Hierarchically Semi-Separable (HSS) matrices, but are in the process of adding several other formats. Figure 7 illustrates the HSS format we currently use as well as two other formats that we are incorporating into the code.
There is theory in the literature that predicts that the off-diagonal blocks of the dense blocks encountered in the sparse elimination have a low-rank off-diagonal structure[3]. However, for the problem being considered here, indefinite Maxwell, the ranks of these off-diagonal blocks increase with the mesh size ($\sim k$ for 3D problems $N = k^3$, or $\sim \log(k)$ for 2D problems $N = k^2$). Theoretically, there is still a clear benefit from exploiting rank-structured blocks, since the size of the rank-structure matrices (size of the top mesh separator) grows as k^2 (3D) or k (2D). However, in practice these ranks become prohibitively large and we do not get good low-rank compression of the fill. Or we get compression, but the additional cost of computing the low-rank compression does not pay-off. This could also be due to our current randomized low-rank compression scheme, which loses accuracy when the singular value decay is relatively slow. We intend to overcome these difficulties by adding two new rank-structured formats to the STRUMPACK preconditioner:

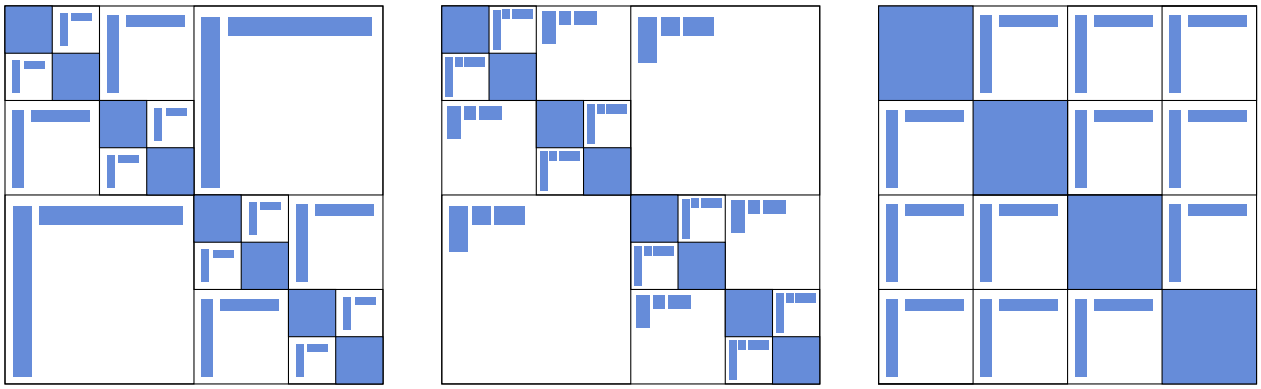
- **BLR** – The block low-rank format allows for a finer partitioning of the dense fill blocks, which allows a better control of the growth of the ranks. We currently have a sequential/threaded code for BLR matrix compression, that is already incorporated into the STRUMPACK preconditioner. See Figures 8 and 9 for benchmark results and a comparison with HSS.

- **HODLR** – The HODLR format is similar to the HSS format, but is simpler. Both HSS and HODLR assume all off-diagonal blocks are admissible, i.e., numerically low-rank. Theoretically, the HSS format has lower asymptotic complexity. However, in practice, we expect to achieve better scalability from the HODLR code. We have a working parallel HODLR code which we are now incorporating into the sparse solver. For the HODLR construction, we also use random sampling, as for HSS. However, in the HODLR setting, we can employ power iteration in the random sampling, which will improve accuracy, especially in cases with slow singular value decay.

In Figure 8 we show convergence using GMRES or BiCGStab using the STRUMPACK preconditioner with either HSS or BLR compression. These tests are on a relatively small problem (setting $\Omega = 2$), since our BLR code is not distributed memory for now.
From Figure 9(right) we see that both the BLR and HSS preconditioners reduce the memory usage and the required number of floating point operations compared to the direct solvers SuperLU and STRUMPACK. However, for higher accuracy, the HSS preconditioner takes too much memory, flops and time. For these higher accuracies, the BLR preconditioner still requires less memory and flops. However, looking at the run-time in Figure 9(left), we see that the direct solvers become

[3] Shiv Chandrasekaran et al. "On the numerical rank of the off-diagonal blocks of Schur complements of discretized elliptic PDEs." SIAM Journal on Matrix Analysis and Applications 31.5 (2010): 2261-2290.

more efficient. This is due to the fact that the (algorithmically simpler) direct solvers reach a higher floating point throughput. We expect better speedups from the BLR preconditioner for larger problems, but we are currently restricted to smaller problems since our BLR code is not distributed memory yet.



HODLR HSS BLR

Figure 7: Illustration of several rank-structured matrix formats. Both HODLR and HSS assume all off-diagonal blocks are of low numerical rank. The Block Low-Rank (BLR) format is somewhat more general. HSS is more complicated than HODLR because it uses what is known as nested-bases: low-rank factors on one level are expressed as linear combinations of the bases on a finer level.

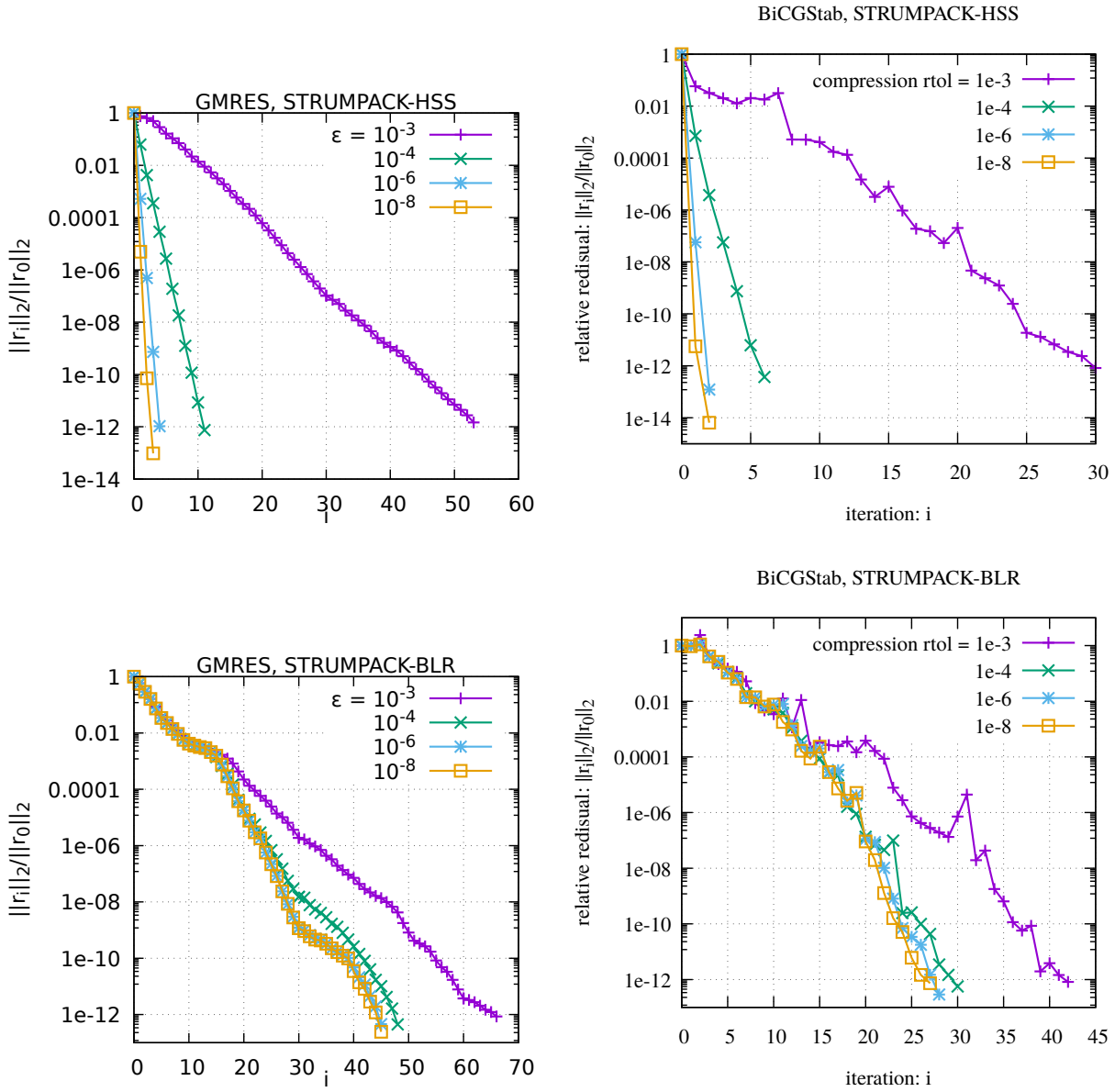


Figure 8: Convergence results for `periodic-cube.mesh`, with $\Omega = 2$, using the STRUMPACK rank-structured preconditioner, with either HSS or BLR compression, for different relative low-rank compression tolerances and for the BiCGStab and GMRES iterative solvers. The matrix has 331,776 rows and 10,948,608 nonzeros. All tests are run sequentially.

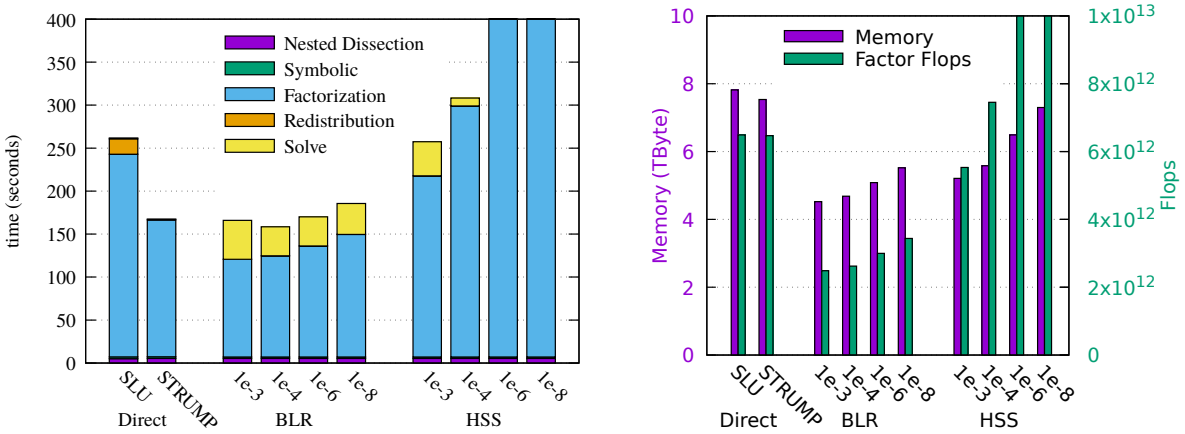


Figure 9: Timings, number of floating point operations and memory usage for `periodic-cube.mesh`, with $\Omega = 2$, using the STRUMPACK rank-structured preconditioner, with either HSS or BLR compression, for different relative low-rank compression tolerances. The absolute low-rank compression tolerance was 10^{-10}. The number of floating point operations shown is for the factorization; those for the solve phase (entire GMRES) are negligible compared to those for the factorization. The matrix has 331,776 rows and 10,948,608 nonzeros. All tests are run sequentially.

1.2 ExaSGD – Optimizing Stochastic Grid Dynamics

ExaSGD uses a parallel nonlinear, nonconvex optimization approach for optimizing large scale power grid planning. It is a multiperiod contingency constrained alternating current optimal power flow problem (`https://confluence.exascaleproject.org/display/ADSE22`). The PIPS-NLP parallel interior-point optimization algorithm is used to solve the NLP problem. The indefinite (saddle-point) linear systems involving Jacobian and Hessian are to be solved at each Newton step required by the interior-point method.

The current algorithm in PIPS uses Schur-complement method (non-overlapping domain decomposition), where each subdomain is solved by a sparse direct solver MA57 (serial) and the Schur-complement system is solved by a dense direct solver Elemental (parallel). We refer to this as the baseline algorithm.

Earlier this year, we held a couple of conference calls with the ExaSGD team (Michel Schanen, Cosmin Petra and Mihai Anitescu). Below is a summary of the problem characteristics:

- PIPS code is in C++. Linear solvers take over 90% of the simulation time.
- PIPS interior point method takes about 50 iterations. The first 25 iterations do not need be accurate for the linear solver, 2-3 digits is sufficient.
- The largest problem may have up to 1 billion dofs, with 10,000 diagonal blocks (subdomains), each of size 100,000. This problem size needs to stay with the Schur-complement solver framework. We agreed that the potential improvements are to replace the serial MA57 and dense Elemental by parallel STRUMPACK/SuperLU sparse solvers as follows.
  - For large problems, the subdomain problem can be solved on multiple compute nodes.

- The Schur complement is sparse, with sparsity pattern shown in Fig. 2 of their paper[4], so sparse solvers would be more efficient.

We also identified an open problem for STRUMPACK – there is a need to develop a new code to compute the inertia of the matrix.
As a first step of collaboration, the ExaSGD team dumped the two typical matrices for us to investigate our solvers performance. We focused on the larger one – the power grid has 118 buses and 64 contingencies. The linear system has about 1.5M dofs; we refer to it as matrix `118_1536`. The following subsections describe our findings.

1.2.1 Solvers performance

Since the current linear system size of 1.5M is in the performance sweet-spot region of direct solvers, we applied our solvers to the entire system, instead of using the Schur-complement method. We first applied STRUMPACK to solve matrix `118_1536`. Using 1536 cores of NERSC Cori-KNL, we observed 6x faster than the baseline algorithm. Next, we performed initial scaling study of SuperLU for this problem, using up to 2025 cores of NERSC Cori-Haswell. The factorization time and the triangular solution time are about 17 seconds and 0.4 seconds, respectively. Since then, we have significantly improved the triangular solution algorithm and now the solve time is reduced to 0.05 seconds, i.e., 7x improvement on 2025 cores.
Similar to the MFEM problem, we identified a bottleneck of fill-reducing ordering using METIS; it takes about 19 seconds, and does not scale. Fortunately, since this problem involves nonlinear iterations, the linear systems of the same sparsity pattern needs to be solved repeatedly, whereas the fill-reducing ordering needs to be done only once. Therefore, its cost can be amortized throughout the nonlinear iterations. Furthermore, we can also use the remedies discussed in Section 2 to reduce the ordering cost.

1.2.2 Computing inertia in STRUMPACK

For a symmetric matrix, the **inertia** $\nu(A)$, is the counts of positive, negative, and zero eigenvalues. In ExaSGD, the inertia is utilized to verify optimal solutions of constrained optimization problems. The standard way of computing the inertia is to perform an LDL^T factorization (symmetric and indefinite), and the counts of positive, negative and zero values on the diagonal matrix D corresponds to the inertia count.
STRUMPACK is designed for unsymmetric factorization which limits its use in optimization problems. In the summer of 2018, we hired a graduate student Jonas Actor of Rice University to investigate the problem. We developed a new algorithm to perform HSS-based LDL^T factorization, which preserves symmetry and the low-rank compression. We extended STRUMPACK so that the factors from STRUMPACK's matrix compression are reused to hierarchically decompose Schur complements, thereby enabling the calculation of inertia in nearly linear time. The algorithm is well tested in Matlab and and the C++ serial code is already implemented in STRUMPACK. Our future work is to parallelize it in STRUMPACK parallelization framework.

[4] Michel Schanen, Francois Gilbert, Cosmin G. Petra, and Mihai Anitescu, "Toward Multiperiod AC-Based Contingency Constrained Optimal Power Flow at Large Scale", The Power Systems Computation Conference, May 11-15, 2018.

2 Conclusions and Future Work

We list several bottlenecks we have identified, and suggest potential remedies.

- **BOTTLENECK:** We identified an issue with SuperLU supernode relaxation when using parallel symbolic factorization, and the parallel fill-reducing ordering ParMETIS.

  Suggested REMEDY: Implement supernode relaxation in the parallel symbolic factorization phase in SuperLU.

- **BOTTLENECK:** Parallel scalability of fill-reducing ordering codes is a real concern. This gets even worse since, for parallel orderings, the quality often degrades. There are also licensing issues with ParMETIS.

  Suggested REMEDIES:

  - Redistribute the graph to a handful of nodes and call ParMETIS on this subset of nodes. This allows to solve larger problems than with sequential METIS, and it does not suffer from the serious slow-downs, and degraded ordering quality, which was observed for ParMETIS on large number of nodes.
  - We have started development of a spectral nested dissection code, with promising initial results. We will continue development of this new ordering code.

- **BOTTLENECK:** Our current rank-structured preconditioner, based on Hierarchically Semi-Separable matrices, is not effective for the numerically difficult indefinite Maxwell problem considered in Section 1.1.

  Suggested REMEDIES:

  - In this report, we have presented promising initial (sequential) results with an alternative to compression using HSS matrices, i.e., Block Low-Rank (BLR) compression. We consider further development of the parallel implementation of the BLR preconditioner as well as off-loading to GPU's. We believe the BLR algorithms is more amenable to GPU parallelization than the HSS code.
  - Apart from HSS and BLR, we are also incorporating Hierarchically Off-Diagonal Low-Rank (HODLR) compression in the STRUMPACK preconditioner. This has theoretically worse complexity than the HSS code, but is much simpler. It has smaller prefactors in the complexity, and it might scale better in parallel.
  - Ultimately, we want to implement butterfly compression[5] in the preconditioner. This will be combined with a hierarchical matrix decomposition such as HODLR, but the low rank factorization is replaced by a multilevel low-rank format (with certain similarities to the fast Fourier transform). This approach seems promising for high frequency problems.

[5] Liu, Yang, Han Guo, and Eric Michielssen. "An HSS matrix-inspired butterfly-based direct solver for analyzing scattering from two-dimensional objects." IEEE Antennas and Wireless Propagation Letters 16 (2017): 1179-1183.