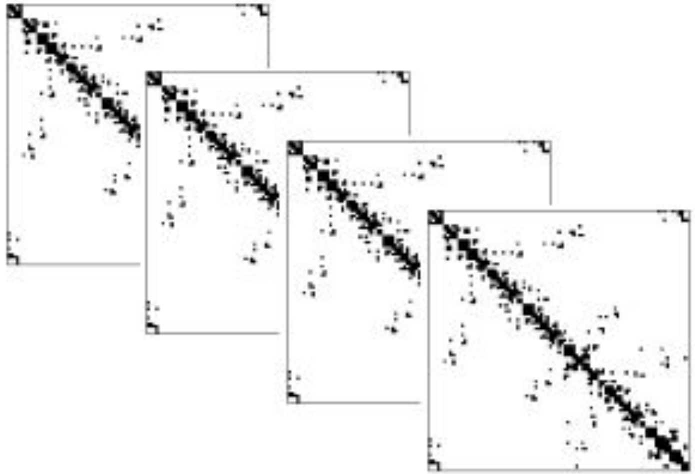
Preconditioners for Batched Iterative Linear Solvers on GPUs

Isha Aggarwal, Pratik Nayak, Aditya Kashi, Hartwig Anzt

What are batched methods ?

- Batching: Related but independent computations that can be scheduled in parallel.
- Are highly suitable for GPUs and processors with many parallel computing units.
- Can maximize utilization of the GPU, due to excellent scalability.



Related work

- Usage in block-Jacobi preconditioners (Anzt. et.al PMAM 17)
- Batched BLAS interface (Dongarra et.al 2016)
- Dense triangular solves on GPUs, DGETRF (Dong et.al 2014)
- Tri-/Penta- diagonal banded solvers on GPUs (Carroll et.al 2021, Gloster et.al 2019, Valero-Lara et.al 2018)

Iterative methods ?

- To solve $Ax = b$ iteratively
- Richardson or similar fixed point methods

$$x_{k+1} = Gx_k + f$$

- Krylov subspace methods

$$b - Ax_m \perp \mathcal{L}_m$$

Different choices of the subspace $\mathcal{L}_m$ give rise to different methods

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\}$$

- Examples: CG, BiCGSTAB, GMRES etc (Saad 2003)

Why batched iterative methods ?

- Most current research and software focuses on dense and direct solvers.
- For medium sized problems, dense and/or direct methods run into memory issues.
- Very high accuracy not usually required. Iterative methods provide tunable accuracy.
- Some applications have matrices with relatively low condition numbers.

Challenges

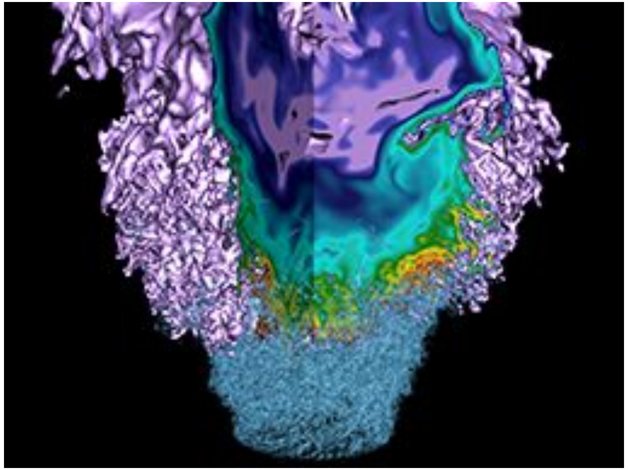
- Memory bound nature of sparse iterative methods.
- Iterative methods usually have a lot of distinct kernels. Overhead of kernel launches can be significant.
- Explosion of parameters for iterative solvers requires attention to interface design.
- Balancing composability and flexibility can be difficult.
- Optimization of sparse matrix storage very important.
- Independent convergence and stopping for each individual linear system.

Opportunities

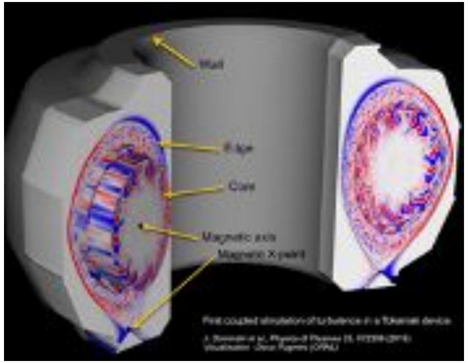
- Relatively cheap computational cost for small to medium sized problems.
- Tunable accuracy can improve overall time to solution.
- Shared sparsity pattern can allow for optimized storage and caching matrices in constant memory.
- Linear system solution inside a non-linear loop can make use of better initial guesses from previous iterations.
- Independent convergence and stopping for each individual linear system.

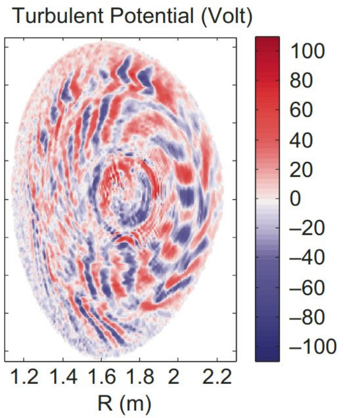
Applications

Combustion simulation: PeleLM from the SUNDIALS suite.



XGC: A fusion plasma simulation using the Gyrokinetic particle in Cell method.





Ginkgo's batched interface: Objectives

- Store one copy of the sparsity pattern and store the different values.
- Provide different Sparse matrix formats to support different sparsity patterns.
- Provide a wide variety of solvers for both symmetric and non-symmetric problems.
- Fuse kernels to maximize cache usage and reduce kernel launch latency.

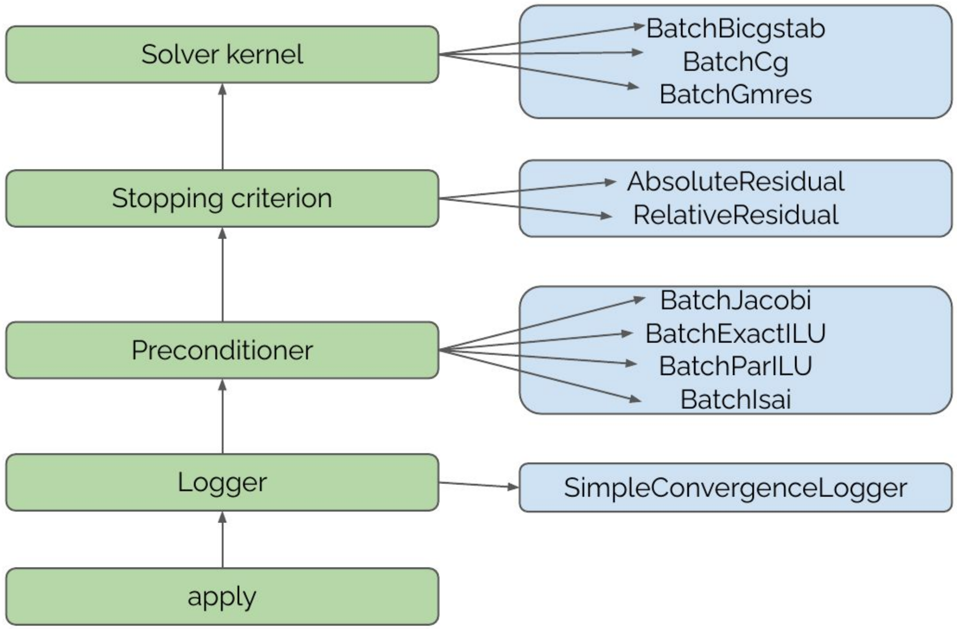
https://github.com/ginkgo-project/ginkgo/tree/batch-develop

Ginkgo's batched interface: Design

- Sparse matrix formats: `BatchCsr` and `BatchEll`
- Iterative solvers: `BatchBicgstab`, `BatchGmres`, `BatchCg`, `BatchIdr` and `BatchRichardson`
- Preconditioners: `BatchJacobi`, `BatchExactILU`, `BatchParILU`, `BatchIsai`
- Template the global apply kernel on logger, stopping criterion, matrix type and preconditioner type.
- Pre-configure dynamic shared memory based on problem size.
- Each problem solved on one thread block (But variants are WIP).

Multi-level dispatch mechanism (host-side)

- Host side dispatch and the solver kernel is templated.
- Matrix format is also templated.
- Allows for easy addition of new features and functionality and eases maintenance.



How does the interface look ?

```
template <typename StopType, typename PrecType,
    typename LogType, typename BatchMatrixType,
    typename ValueType>
__global__ void apply_kernel(int padded_length,
    const StorageConf config, int max_iter,
    remove_complex<ValueType> tol,
    LogType logger, PrecType preconditioner,
    const BatchMatrixType a,
    const ValueType *__restrict__ b,
    ValueType *__restrict__ x,
    ValueType *__restrict__ workspace)
```

How does the interface look ?

```
apply_kernel<stop::SimpleRelResidual<ValueType>>
    <<<nbatch, block_size, shared_size>>>(
        shared_gap, config, max_its, residual_tol,
        logger, precond, a,
        b.values, x.values, workspace);
```

Optimization: Automatic shared memory config

- Red objects: Intermediate vectors in SpMV: High priority
- Blue objects: Other vectors: Low priority
- Green objects: Constant matrices or vectors (In constant cache)

$\boldsymbol{r} \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \hat{\boldsymbol{r}} \leftarrow \boldsymbol{r}, \boldsymbol{p} \leftarrow 0, \boldsymbol{v} \leftarrow 0$
$\rho' \leftarrow 1, \omega \leftarrow 1, \alpha \leftarrow 1$
for $i < N_{iter}$ **do**
    if $\|\boldsymbol{r}\| < \tau$ **then**
        Break
    end if
    $\rho \leftarrow \boldsymbol{r} \cdot \boldsymbol{r}'$
    $\beta \leftarrow \frac{\rho' \alpha}{\rho \omega}$
    $\boldsymbol{p} \leftarrow \boldsymbol{r} + \beta(\boldsymbol{p} - \omega \boldsymbol{v})$
    $\hat{\boldsymbol{p}} \leftarrow \text{PRECOND}(\boldsymbol{p})$
    $\boldsymbol{v} \leftarrow \boldsymbol{A}\hat{\boldsymbol{p}}$
    $\alpha \leftarrow \frac{\rho}{\hat{\boldsymbol{r}} \cdot \boldsymbol{v}}$
    $\boldsymbol{s} \leftarrow \boldsymbol{r} - \alpha \boldsymbol{v}$
    if $\|\boldsymbol{s}\| < \tau$ **then**
        $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \hat{\boldsymbol{p}}$
        Break
    end if
    $\hat{\boldsymbol{s}} \leftarrow \text{PRECOND}(\boldsymbol{s})$
    $\boldsymbol{t} \leftarrow \boldsymbol{A}\hat{\boldsymbol{s}}$
    $\omega \leftarrow \frac{\boldsymbol{t} \cdot \boldsymbol{s}}{\boldsymbol{t} \cdot \boldsymbol{t}}$
    $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \hat{\boldsymbol{p}} + \omega \hat{\boldsymbol{s}}$
    $\boldsymbol{r} \leftarrow \boldsymbol{s} - \omega \boldsymbol{t}$
    $\rho' \leftarrow \rho$
end for

An Exact ILU(0) preconditioner

- In place factorization
- Updated in parallel over batch entries.
- One warp per row for coalesced access.
- Store the current row's elements in shared memory.

Algorithm 1 The batched Exact ILU(0) algorithm

```
INPUT: A
OUTPUT: Factorized (in-place) A ≈ LU
N ← num_rows
for b = 0 to num_batch_entries − 1 do
    for i = 0 to N − 1 do
        for k = i + 1 to N − 1 do
            row ← 0
            if (k, i) ∈ spy(A_b) then
                A_b(k, i) ← A_b(k, i)/A(i, i)
                row ← A_b(k, i)
            end if
            for c = i + 1 to N − 1 and (k, c) ∈ spy(A_b) do
                col ← 0
                if (i, c) ∈ spy(A) then
                    col ← A_b(i, c)
                end if
                A_b(k, c) ← A_b(k, c) − (row * col)
            end for
        end for
    end for
end for
```

Batched ISAI algorithm

- Store sparsity pattern of the matrix or of the powers in cache.
- Loop over all rows in entire batch.
- Assign subwarps to row.
- Extract matrix values into a vector and compute a direct solution.
- Triangular solve or general solve depending on type of matrix assembled.

Algorithm 3 The batched ISAI algorithm.

INPUT: A, k
OUTPUT: $\hat{A}$
$S \leftarrow spy(|A_0|^k)$
$spy(\hat{A}) \leftarrow S$
for $i = 0$ **to** $num\_rows - 1$ **do**
  $T_i \leftarrow find\_non\_zero\_locations(\hat{A}_0(i,:))$
  $Size_i \leftarrow length(T_i)$
  $M_i \leftarrow generate\_pattern(A_0(T_i, T_i))$
  $R_i \leftarrow find\_location\_one(I(i, T_i))$
  for $v = 0$ **to** $num\_batch\_entries - 1$ **do**
    $\hat{A_v}(i, T_i) * A_v(M_i) = get\_rhs(Size_i, R_i)$
  end for
end for

Batched triangular solvers

- Used in application of the preconditioners or for ILU based ISAI generation.
- Symbolic phase is sequential and computed on the host for one sparsity pattern.
- A busy wait based implementation inside each thread block.
- Avoids synchronization problems due to dependencies.

Experimental setup and test cases

- 3 test cases.
  - Scaling with a 3 point Laplacian stencil problem
  - General matrices from Suitesparse.
  - Practical application problem from PeleLM
- On the HoreKa machine at the Karlsruhe Institute of Technology.
  - Each node has 4 A100 (40 GB) GPUs with 2 Intel Xeon Platinum 8368
  - Software setup: CUDA 11.4 and gcc-10

Test cases

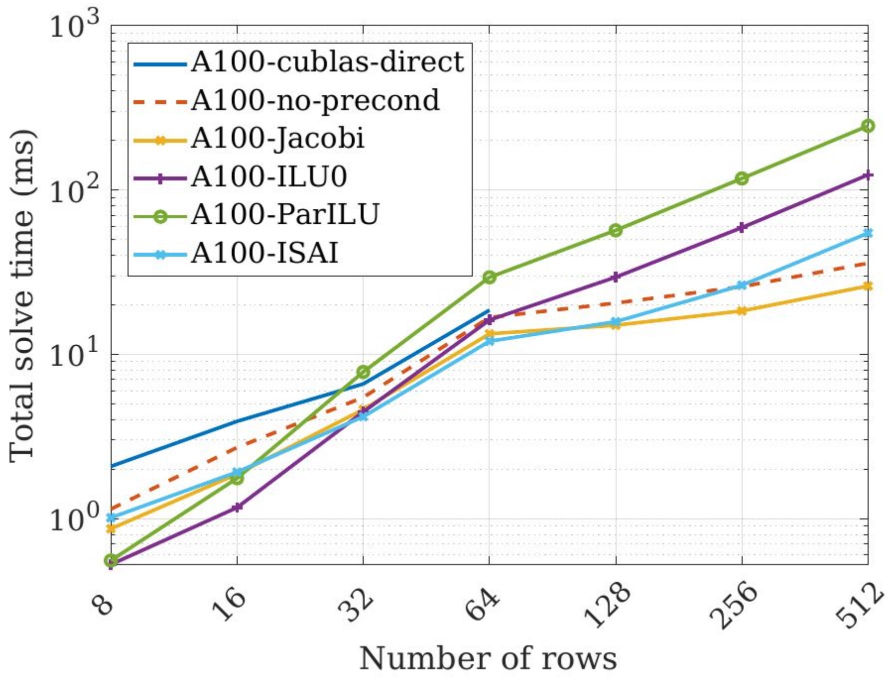
	size	nonzeros	*No Precond*	*Jacobi*	*ILU(0)*	*ParILU*	*ISAI*
1D Laplace							
3pt-stencil-64	64	190	16	11	1	1	6
Suitesparse							
LFAT5	14	46	80	33	7	8	16
bcsstm02	66	66	11	1	1	1	1
LF10	18	82	351	–	38	34	–
Trefethen_20	20	158	19	8	5	5	6
pivtol	102	306	16	13	2	2	7
bfwb62	62	342	30	15	6	6	9
olm100	100	396	–	–	36	98	26
bcsstk22	138	696	493	229	43	42	95
cage6	93	785	–	12	4	4	7
ck104	104	992	112	118	13	15	164
494_bus	494	1666	–	–	81	81	–
mesh3em5	289	1889	14	13	1	1	10
mhdb416	416	2312	–	37	2	2	41
bcsstk05	153	2423	325	124	32	32	149
steam1	240	3762	–	–	3	3	–
PeleLM							
isooctane	144	6135	–	38	3	4	–

Significant reduction in iteration count

Sophisticated preconditioners necessary for some problems

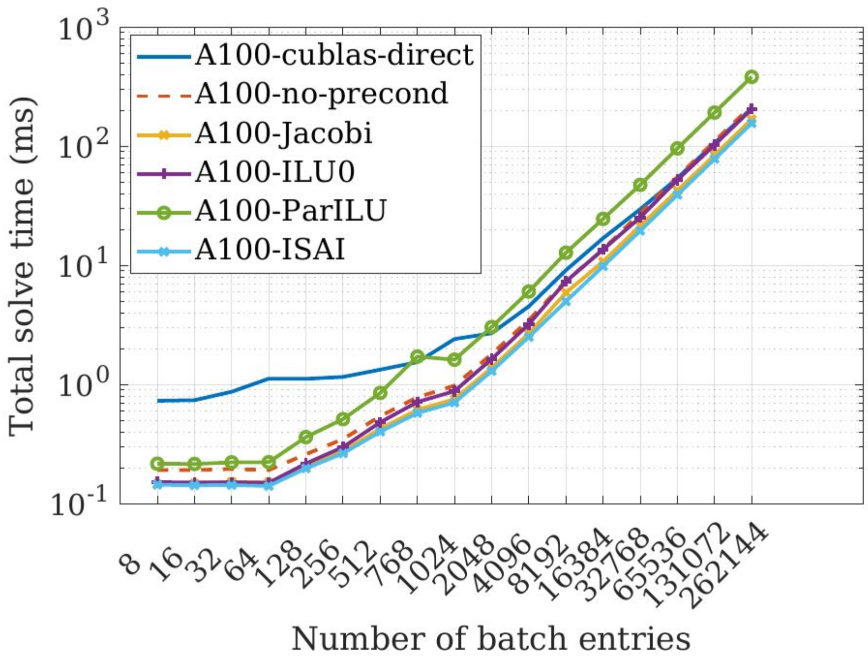
Scaling with a 3 point stencil

- Increase size of individual batch entries, fixed number of batch entries (20000)
- Dense direct method does not scale beyond 64 rows due to out of memory issues

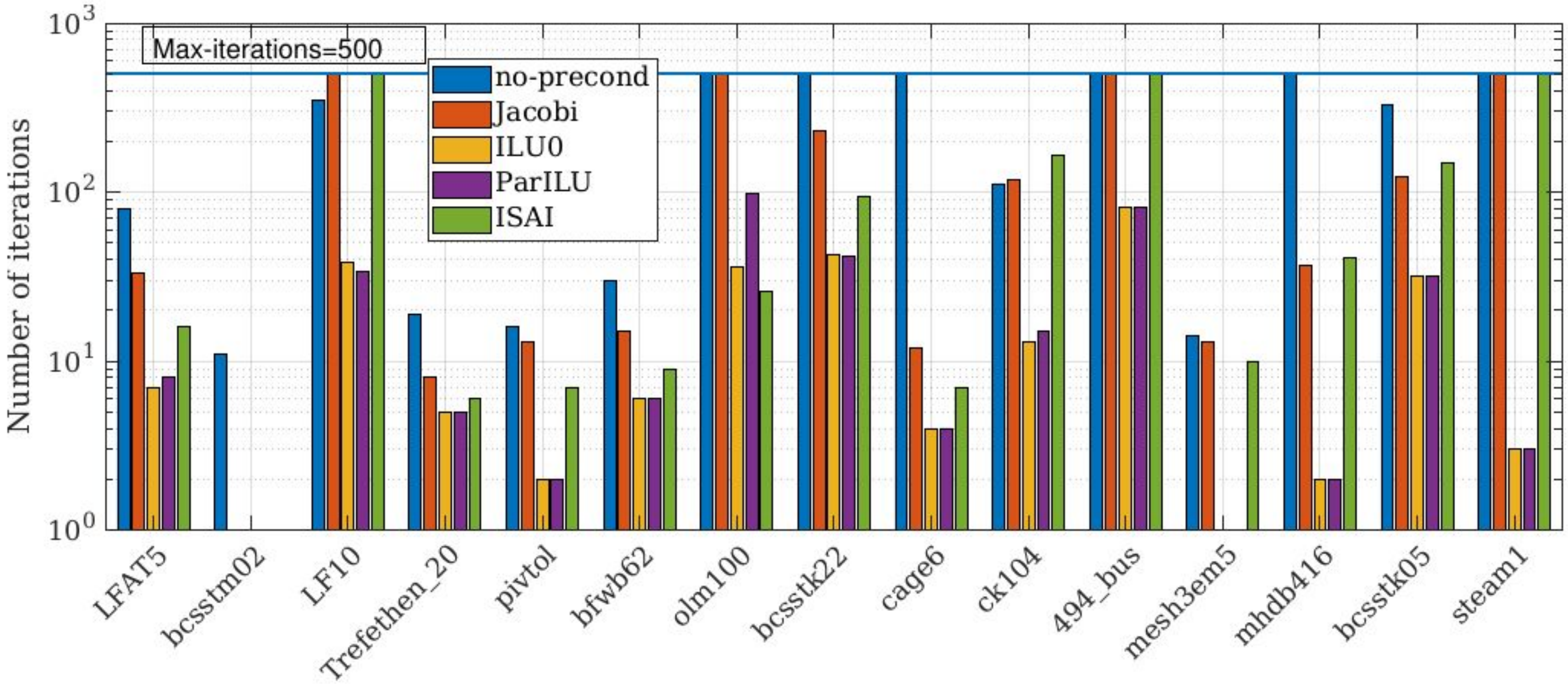


Scaling with a 3 point stencil

- Laplacian 3-pt stencil, each entry has 64 rows.
- Upto 40x speedup for modest number of batch entries
- Upto 1.5x speedup for very large number of batch entries.

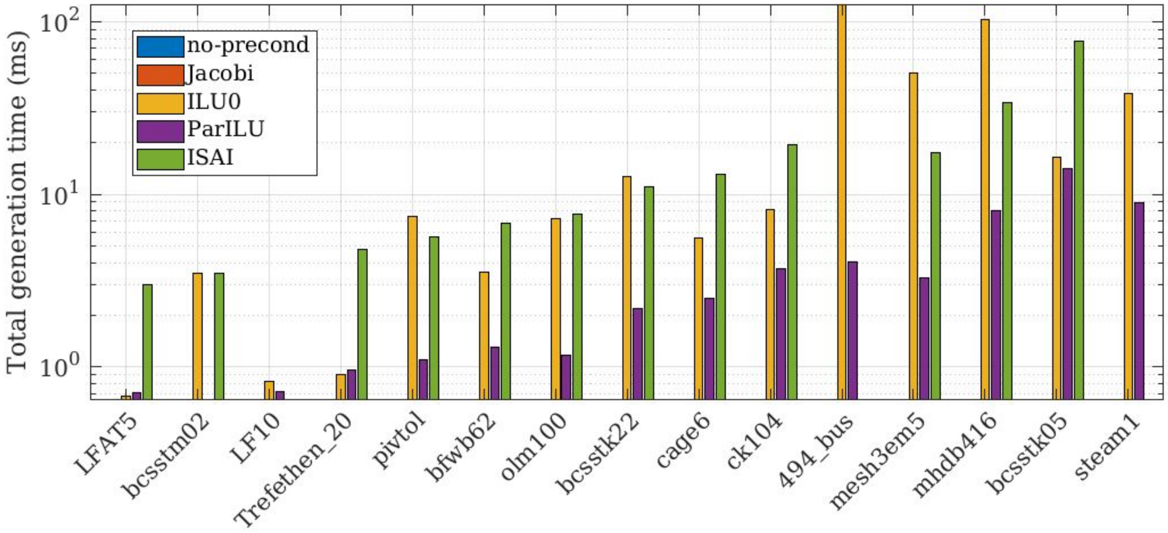


Iteration counts



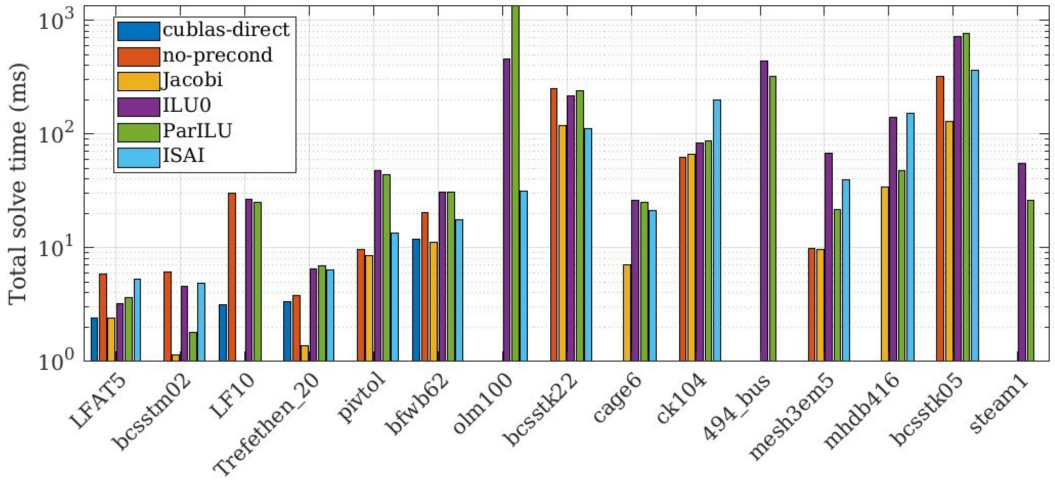
Time for generation

- Generation of the preconditioner can be significant, even if a one time cost.
- ILU(0) is expensive to generate.
- ParILU is cheaper can be less effective.
- ISAI can be a compromise between cost and effectiveness.



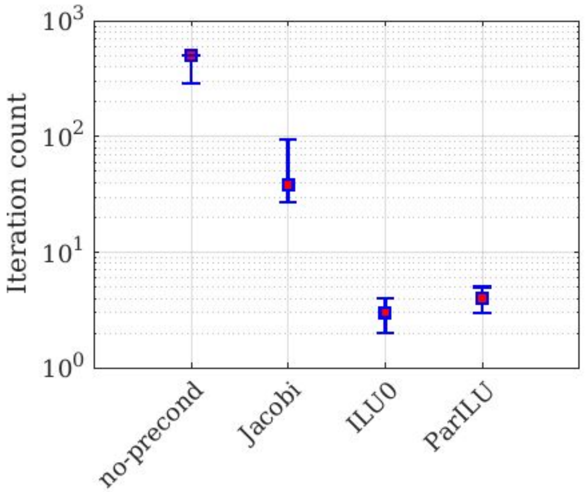
Total solve time: generation (once) + application

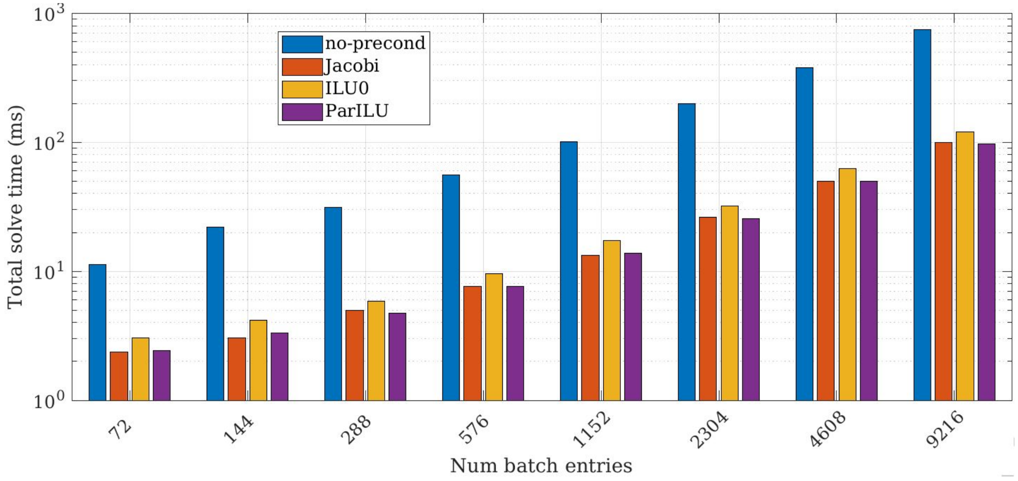
- ILU(0) is the most robust and enables solution for all problems.
- ParILU can win in some cases due to cheaper generation.
- Scalar Jacobi can still be very effective despite large number of iterations.



Are preconditioners useful ? The Isooctane problem

- Some variation in iteration counts in problems in a batch.
- Preconditioners can significantly reduce the iteration count.





Conclusion and future work

- Batched iterative solvers have shown to be effective in a variety of cases.
- Batched preconditioners are necessary for more complex problems and can help in improving the performance further.
- ISAI currently only works for problems with num nonzeros per row < 32.
- ILU with ISAI, ISAI(k) and Block Jacobi preconditioners are work in progress and have shown promise

Thank you!
Any questions ?

Ginkgo

https://github.com/ginkgo-project/ginkgo

Why not Block Diagonal assembly ?

- Need to wait for slowest problem; independent stopping is difficult.
- Eigenvalues of the monolithic problems union of the eigenvalues of the individual problems.

$$\mathbf{A} = \begin{bmatrix} A_1 & 0 & \cdots & 0 \\ 0 & A_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_K \end{bmatrix}.$$