August 24, 2011

Solving Large Scale Matrix Equations on Multicore-CPUs

Martin Köhler
joint work with
Peter Benner and Jens Saak

Young Researchers Minisymposium **YR6**:
Parallel Computing in Numerical Linear Algebra

MAX PLANCK INSTITUTE
FOR DYNAMICS OF COMPLEX
TECHNICAL SYSTEMS
MAGDEBURG

Outline

Motivation

Motivation

Solving Large Scale Lyapunov Equations: LRCF - ADI e.g.,[Benner/Li/Penzl '08, Saak '09]

Consider: $FXE^T + EXF^T = -GG^T \qquad E, F \in \mathbb{R}^{n\times n}, G \in \mathbb{R}^{n\times p}$

Task: Find $Z \in \mathbb{K}^{n,n_z}$, such that $n_z \ll n$ and $X \approx ZZ^H$

Motivation

Solving Large Scale Lyapunov Equations: LRCF - ADI e.g.,[Benner/Li/Penzl '08, Saak '09]

Consider: $FXE^T + EXF^T = -GG^T \qquad E, F \in \mathbb{R}^{n\times n}, G \in \mathbb{R}^{n\times p}$

Task: Find $Z \in \mathbb{K}^{n,n_z}$, such that $n_z \ll n$ and $X \approx ZZ^H$

Algorithm

$$V_1 = \sqrt{-2p_1}(F + p_1 E)^{-1} G, \qquad Z_1 = V_1$$

$$V_i = \frac{\sqrt{p_i}}{\sqrt{p_{i-1}}}\left[I - (p_i + \overline{p_{i-1}})(F + p_i E)^{-1}\right] EV_{i-1}, \qquad Z_i = [Z_{i-1} \quad V_i]$$

For certain shift parameters $\{p_1, ..., p_J\} \subset \mathbb{C}_{<0}$.

Stop if

- $\|V_i V_i^H\|$ is small, or
- $\|FZ_i Z_i^H E^T + EZ_i Z_i^H F^T + GG^T\|$ is small.

Motivation

Solving Large Scale Lyapunov Equations: LRCF - ADI

Algorithm 1 Low-rank Cholesky factor ADI iteration (LRCF-ADI)
[Penzl '97/'00, Li/White '99/'02, Benner/Li/Penzl '99/'08]

Input: F, G defining $FX + XF^T = -GG^T$ and shifts $\{p_1, \ldots, p_{i_{max}}\}$
Output: $Z = Z_{i_{max}} \in \mathbb{C}^{n \times t_{i_{max}}}$, such that $ZZ^H \approx X$

1: For V_1 solve $(F + p_1 I) V_1 = \sqrt{-2 \operatorname{Re}(p_1)} G$
2: $Z_1 = V_1$
3: **for** $i = 2, 3, \ldots, i_{max}$ **do**
4: For $\tilde{V}$ solve $(F + p_i I)\tilde{V} = V_{i-1}$
5: $V_i = \sqrt{\operatorname{Re}(p_i)/\operatorname{Re}(p_{i-1})} \left(V_{i-1} - (p_i + \overline{p_{i-1}})\tilde{V} \right)$
6: $Z_i = [Z_{i-1} \;\; V_i]$
7: **end for**

Motivation

Solving Large Scale Lyapunov Equations: LRCF - ADI

Algorithm 1 General. Low-rank Cholesky factor ADI iteration (G-LRCF-ADI)
[Benner '04, Benner/Saak. '09, Saak. '09]

Input: E, F, G defining $FXE^T + EXF^T = -GG^T$ and shifts $\{p_1, \ldots, p_{i_{max}}\}$
Output: $Z = Z_{i_{max}} \in \mathbb{C}^{n \times t_{i_{max}}}$, such that $ZZ^H \approx X$

1: For V_1 solve $(F + p_1 E)\, V_1 = \sqrt{-2\,\mathrm{Re}\,(p_1)}\, G$
2: $Z_1 = V_1$
3: **for** $i = 2, 3, \ldots, i_{max}$ **do**
4: For $\tilde{V}$ solve $(F + p_i E)\tilde{V} = E V_{i-1}$
5: $V_i = \sqrt{\mathrm{Re}\,(p_i)/\,\mathrm{Re}\,(p_{i-1})}\left(V_{i-1} - (p_i + \overline{p_{i-1}})\tilde{V}\right)$
6: $Z_i = [Z_{i-1} \;\; V_i]$
7: **end for**

Motivation

Solving Large Scale Lyapunov Equations: LRCF-ADI

- Dense operations "parallelized" using optimized BLAS implementation.
- Sparse matrix vector product parallelized using OpenMP.

Algorithm 1 Low-rank Cholesky factor ADI iteration (G-LRCF-ADI)
[Benner '04, Benner/Saak '09, Saak '09]

Input: E, F, G defining $FXE^T + EXF^T = -GG^T$ and shifts $\{p_1, \ldots, p_{i_{max}}\}$
Output: $Z = Z_{i_{max}} \in \mathbb{C}^{n \times t_{i_{max}}}$, such that $ZZ^H \approx X$

1: For V_1 solve $(F + p_1 E)\, V_1 = \sqrt{-2\,\mathrm{Re}\,(p_1)}\, G$
2: $Z_1 = V_1$
3: **for** $i = 2, 3, \ldots, i_{max}$ **do**
4: For $\tilde{V}$ solve $(F + p_i E)\tilde{V} = EV_{i-1}$
5: $V_i = \sqrt{\mathrm{Re}\,(p_i)/\mathrm{Re}\,(p_{i-1})}\left(V_{i-1} - (p_i + \overline{p_{i-1}})\tilde{V}\right)$
6: $Z_i = [Z_{i-1} \;\; V_i]$
7: **end for**

Motivation

Solving Large Scale Lyapunov Equations: LRCF - ADI

Algorithm 1 General. Low-rank Cholesky factor ADI iteration (G-LRCF-ADI) [Benner '04, Benner/Saak. '09, Saak. '09]

Input: E, F, G defining $FXE^T + EXF^T = -GG^T$ and shifts $\{p_1, \ldots, p_{i_{max}}\}$
Output: $Z = Z_{i_{max}} \in \mathbb{C}^{n \times t_{i_{max}}}$, such that $ZZ^H \approx X$

1: For V_1 solve $(F + p_1 E)\, V_1 \leftarrow \sqrt{-2 \operatorname{Re}(p_1)}\, G$
2: $Z_1 = V_1$
3: **for** $i = 2, 3, \ldots, i_{max}$ **do**
4: For $\tilde{V}$ solve $(F + p_i E)\, \tilde{V} \leftarrow E V_{i-1}$
5: $V_i = \sqrt{\operatorname{Re}(p_i)/\operatorname{Re}(p_{i-1})} \left(V_{i-1} - (p_i + \overline{p_{i-1}})\, \tilde{V} \right)$
6: $Z_i = [Z_{i-1} \;\; V_i]$
7: **end for**

Parallelization of sparse direct solvers is not trivial.

Motivation

Solving Sparse-Dense Sylvester Equations

[Benner/K./Saak. '11]

Consider: $AX + XH = M \qquad A \in \mathbb{R}^{n\times n}, H \in \mathbb{R}^{m\times m}, M \in \mathbb{R}^{n\times m}$
with A sparse and large, H small and dense.

Task: Find X with a direct solver without transforming A.

Motivation

Solving Sparse-Dense Sylvester Equations [Benner/K./Saak. '11]

Consider: $AX + XH = M \qquad A \in \mathbb{R}^{n\times n}, H \in \mathbb{R}^{m\times m}, M \in \mathbb{R}^{n\times m}$
with A sparse and large, H small and dense.

Task: Find X with a direct solver without transforming A.

Algorithm

1. Compute the Schur decomposition $USU^T = H$.
2. Transform equation to $AXU + XUS + MU = 0$.
3. Forward/Backward substitution for the columns of MU.

Motivation

Solving Sparse-Dense Sylvester Equations [Benner/K./Saak. '11]

Consider: $AX + XH = M \qquad A \in \mathbb{R}^{n\times n}, H \in \mathbb{R}^{m\times m}, M \in \mathbb{R}^{n\times m}$
with A sparse and large, H small and dense.

Task: Find X with a direct solver without transforming A.

Algorithm

1. Compute the Schur decomposition $USU^T = H$.
2. Transform equation to $AXU + XUS + MU = 0$.
3. Forward/Backward substitution for the columns of MU.

$\rightarrow$ every substitution step needs one solve with $(A + s_{ii} I)$

Motivation

Solving Sparse-Dense Sylvester Equations

[Benner/K./Saak. '11]

Consider: $AX + XH = M \qquad A \in \mathbb{R}^{n\times n}, H \in \mathbb{R}^{m\times m}, M \in \mathbb{R}^{n\times m}$
with A sparse and large, H small and dense.

Task: Find X with a direct solver without transforming A.

Algorithm

1. Compute the Schur decomposition $USU^T = H$.
2. Transform equation to $AXU + XUS + MU = 0$.
3. Forward/Backward substitution for the columns of MU.

$\rightarrow$ every substitution step needs one solve with $(A + s_{ii} I)$

Similar algorithm for $AXF + EXH + M = 0$:

- Generalized Schur decomposition $(QTZ^T, QSZ^T) = (F, H)$.
- Solve $(t_{ii} A + s_{ii} E)$ in every step.

Motivation

Solving Sparse-Dense Sylvester Equations

[Benner/K./Saak '11]

Algorithm 2 Solution of the Sparse-Dense Sylvester Equation

Input: A, H, M defining $AX + XH + M = 0$

Output: Solution $X \in \mathbb{R}^{n \times r}$

1: Compute Schur decomposition $USU^T = H$.
2: $\tilde{M} = MU$
3: **for** $j = 1, \ldots, r$ **do**
4: $\quad \hat{M} = -\tilde{M}_j - \sum_{i=1}^{j-1} S_{ij} \tilde{X}_i$
5: $\quad$ Solve $(A + s_{jj} I)\tilde{X}_j = \hat{M}$.
6: **end for**
7: $X = \tilde{X} U$

Motivation

Solving Sparse-Dense Sylvester Equations

[Benner/K./Saak '11]

Algorithm 2 Solution of the Sparse-Dense Sylvester Equation

Input: A, H, M defining $AX + XH + M = 0$

Output: Solution $X \in \mathbb{R}^{n \times r}$

1: Compute Schur decomposition $USU^T = H$.
2: $\tilde{M} = MU$
3: **for** $j = 1, \ldots, r$ **do**
4: $\quad \hat{M} = -\tilde{M}_j - \sum_{i=1}^{j-1} S_{ij} \tilde{X}_i$
5: $\quad$ Solve $(A + S_{jj} I)\tilde{X}_j = \hat{M}$.
6: **end for**
7: $X = \tilde{X} U$

Dense parallelization done with an optimized BLAS.

Motivation

Solving Sparse-Dense Sylvester Equations

[Benner/K./Saak '11]

Algorithm 2 Solution of the Sparse-Dense Sylvester Equation

Input: A, H, M defining $AX + XH + M = 0$

Output: Solution $X \in \mathbb{R}^{n \times r}$

1: Compute Schur decomposition $USU^T = H$.
2: $\tilde{M} = MU$
3: **for** $j = 1, \ldots, r$ **do**
4: $\quad \hat{M} = -\tilde{M}_j - \sum_{i=1}^{j-1} S_{ij} \tilde{X}_i$
5: $\quad$ Solve $(A + S_{jj} I)\tilde{X}_j = \hat{M}$.
6: **end for**
7: $X = \tilde{X} U$

Same crucial linear system like in LRCF-ADI → Same problems.

Motivation

Other Applications for $(A + p_i I)x = b$

- Dominant Pole Algorithm, for instance [Rommes '07]
- Iterative Rational Krylov Algorithm [Gugercin et. al. '08]
- Evaluation of a transfer function of a SISO or MIMO system.
- ...

Motivation

Other Applications for $(A + p_i I)x = b$

- Dominant Pole Algorithm, for instance [Rommes '07]
- Iterative Rational Krylov Algorithm [Gugercin et. al. '08]
- Evaluation of a transfer function of a SISO or MIMO system.
- ...

The solution of $(A + p_i I)x = b$, respectively $(A + p_{ii} E)x = b$, is a key ingredient for many algorithms.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Preparation - Problems Decomposing $LU = (A + p_i I)$

- Large memory consumption:
  - Hold r decompositions in memory.
  - Example: 16 decomposition on a 64bit machine:

	N	size of L+U in MB	16 LUs in MB
FDM	10 000	6.45	103.20
FDM	90 000	90.75	1 452.00
FDM	250 000	285.10	4 561.30
FDM	562 500	718.00	11 488.00
FDM	1 000 000	1 379.00	22 064.00
Rail	79 841	54.11	865.76
Filter3D	106 437	888.05	14 208.80

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Preparation - Problems Decomposing $LU = (A + p_i I)$

- Large memory consumption:
  - Hold r decompositions in memory.
  - Example: 16 decomposition on a 64bit machine:

	N	size of L+U in MB	16 LUs in MB
FDM	10 000	6.45	103.20
FDM	90 000	90.75	1 452.00
FDM	250 000	285.10	4 561.30
FDM	562 500	718.00	11 488.00
FDM	1 000 000	1 379.00	22 064.00
Rail	79 841	54.11	865.76
Filter3D	106 437	888.05	14 208.80

- Impracticable on non HPC machines.
- Too much memory traffic.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Preparation - Problems Decomposing $LU = (A + p_i I)$

- Large memory consumption
- Parallelization of one LU decomposition is difficult:
  - Speedup and scaling depends on the structure of the matrix.
  - Requires a large overhead during the deep analysis of the matrix.
  - Scales possibly not on many core systems.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Preparation - Problems Decomposing $LU = (A + p_i I)$

- Large memory consumption
- Parallelization of one LU decomposition is difficult
- Many information, like reordering or the symbolic analysis, are computed too often.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Preparation - Problems Decomposing $LU = (A + p_i I)$

- Large memory consumption
- Parallelization of one LU decomposition is difficult
- Many information, like reordering or the symbolic analysis, are computed too often.

Our jobs:

- Reduce the memory usage.
- Find another way to parallelize it.
- Reuse information.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Derivation of the Algorithm

Information from $LU = A$

- The pattern of L and U including the number of nonzero elements.
- Sizes of all data structures.
- Fill-in reducing reordering.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Derivation of the Algorithm

Information from $LU = A$

- The pattern of L and U including the number of nonzero elements.
- Sizes of all data structures.
- Fill-in reducing reordering.

Remark: All entries of the pattern must be stored, even if they are zero.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Derivation of the Algorithm

Information from $LU = A$

- The pattern of L and U including the number of nonzero elements.
- Sizes of all data structures.
- Fill-in reducing reordering.

Remark: All entries of the pattern must be stored, even if they are zero.

Definition

Let $A \in \mathbb{R}^{n \times m}$ be a matrix. We call the set

$$\mathcal{P}(A) = \{(i,j) \mid A_{i,j} \neq 0\}$$

the **pattern** of A. Furthermore we define

$$\mathcal{P}_R(A,i) = \{j \mid A_{i,j} \neq 0\}$$

as the **pattern of the i-th row** of A.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Derivation of the Algorithm

Compute $L_i U_i = (A + p_i I)$ with knowledge of $LU = A$.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Derivation of the Algorithm

Compute $L_i U_i = (A + p_i I)$ with knowledge of $LU = A$.

Key oberservation

If $\mathcal{P}(A + p_i I) = \mathcal{P}(A)$ holds:

- $\mathcal{P}(L_i) = \mathcal{P}(L)$ and $\mathcal{P}(U_i) = \mathcal{P}(U)$. [Gilbert '94]

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Derivation of the Algorithm

Compute $L_i U_i = (A + p_i I)$ with knowledge of $LU = A$.

Key oberservation

If $\mathcal{P}(A + p_i I) = \mathcal{P}(A)$ holds:

- $\mathcal{P}(L_i) = \mathcal{P}(L)$ and $\mathcal{P}(U_i) = \mathcal{P}(U)$. [Gilbert '94]

- Symbolic analysis can be skipped.
- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ are invariant w.r.t. to the shift parameter.
- All memory allocations can be done in one step, i.e. no reallocations needed.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Derivation of the Algorithm

Compute $L_i U_i = (A + p_i I)$ with knowledge of $LU = A$.

Key oberservation

If $\mathcal{P}(A + p_i I) = \mathcal{P}(A)$ holds:

- $\mathcal{P}(L_i) = \mathcal{P}(L)$ and $\mathcal{P}(U_i) = \mathcal{P}(U)$. [Gilbert '94]

- Symbolic analysis can be skipped.
- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ are invariant w.r.t. to the shift parameter.
- All memory allocations can be done in one step, i.e. no reallocations needed.

$\rightarrow$ Use $\mathcal{P}(L)$ and $\mathcal{P}(U)$ to compute $L_i U_i = (A + p_i I)$.
$\rightarrow$ Only hold $\mathcal{P}(L)$ and $\mathcal{P}(U)$ one time in memory.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Derivation of the Algorithm - Resulting Algorithm

Algorithm 3 Pattern-Reuse for $\tilde{L}\tilde{U} = \tilde{A}$

Input: $\tilde{A} := A + pI$, $\mathcal{P}(L)$ and $\mathcal{P}(U)$ with $LU = A$ and $\mathcal{P}(A) = \mathcal{P}(\tilde{A})$
Output: $\tilde{L}$, $\tilde{U}$ with $\tilde{L}\tilde{U} = \tilde{A}$

1: $\tilde{U}(1,:) = \tilde{A}(1,:)$
2: **for** $i = 2, \ldots, n$ **do**
3: $\quad w = \tilde{A}(i,:)$
4: $\quad$ **for all** $j \in \mathcal{P}_R(L,i)$ ordered **do**
5: $\quad\quad \tilde{L}(i,j) = \alpha = w(j)/\tilde{U}(j,j)$
6: $\quad\quad w = w - \alpha \cdot \tilde{U}(j,:)$
7: $\quad$ **end for**
8: $\quad$ **for all** $j \in \mathcal{P}_R(U,i)$ **do**
9: $\quad\quad \tilde{U}(i,j) = w(j)$
10: $\quad$ **end for**
11: **end for**

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Derivation of the Algorithm - Resulting Algorithm

Algorithm 3 Pattern-Reuse for $\tilde{L}\tilde{U} = \tilde{A}$

Input: $\tilde{A} := A + pE$, $\mathcal{P}(L)$ and $\mathcal{P}(U)$ with $LU = A + E$ and $\mathcal{P}(A) = \mathcal{P}(\tilde{A})$
Output: $\tilde{L}$, $\tilde{U}$ with $\tilde{L}\tilde{U} = \tilde{A}$

1: $\tilde{U}(1,:) = \tilde{A}(1,:)$
2: **for** $i = 2, \ldots, n$ **do**
3: $\quad w = \tilde{A}(i,:)$
4: $\quad$ **for all** $j \in \mathcal{P}_R(L,i)$ ordered **do**
5: $\quad\quad \tilde{L}(i,j) = \alpha = w(j)/\tilde{U}(j,j)$
6: $\quad\quad w = w - \alpha \cdot \tilde{U}(j,:)$
7: $\quad$ **end for**
8: $\quad$ **for all** $j \in \mathcal{P}_R(U,i)$ **do**
9: $\quad\quad \tilde{U}(i,j) = w(j)$
10: $\quad$ **end for**
11: **end for**

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Parallelization and Efficiency Improving

Observation:

- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ are the same for any p_i.
- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ accessed read only $\rightarrow$ no race conditions.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Parallelization and Efficiency Improving

Observation:

- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ are the same for any p_i.
- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ accessed read only $\rightarrow$ no race conditions.

Realization: Use OpenMP to parallelize the loop over all p_i.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Parallelization and Efficiency Improving

Observation:

- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ are the same for any p_i.
- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ accessed read only $\rightarrow$ no race conditions.

Realization: Use OpenMP to parallelize the loop over all p_i.

Observation: If the set $\{p_i\}_{i=1}^r$ is closed w.r.t to complex conjugation, we can save LU decompositions. For $p_{i+1} = \bar{p}_i$ use $L_{i+1} = \bar{L}_i$ and $U_{i+1} = \bar{U}_i$.
$\rightarrow$ Save up to the half of the decompositions.
$\rightarrow$ But destroys easy OpenMP parallelization.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Parallelization and Efficiency Improving

Observation:

- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ are the same for any p_i.
- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ accessed read only $\rightarrow$ no race conditions.

Realization: Use OpenMP to parallelize the loop over all p_i.

Observation: If the set $\{p_i\}_{i=1}^{r}$ is closed w.r.t to complex conjugation, we can save LU decompositions. For $p_{i+1} = \bar{p}_i$ use $L_{i+1} = \bar{L}_i$ and $U_{i+1} = \bar{U}_i$.
$\rightarrow$ Save up to the half of the decompositions.
$\rightarrow$ But destroys easy OpenMP parallelization.

Solution:
$\rightarrow$ Add a preprocessing to the parallelized loop.
$\rightarrow$ Or use a thread pool (build with PThreads,...).

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Parallelization and Efficiency Improving

Observation:

- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ are the same for any p_i.
- $\mathcal{P}(L)$ and $\mathcal{P}(U)$ accessed read only $\rightarrow$ no race conditions.

Realization: Use OpenMP to parallelize the loop over all p_i.

Observation: If the set $\{p_i\}_{i=1}^{r}$ is closed w.r.t to complex conjugation, we can save LU decompositions. For $p_{i+1} = \bar{p}_i$ use $L_{i+1} = \bar{L}_i$ and $U_{i+1} = \bar{U}_i$.

$\rightarrow$ Save

$\rightarrow$ But d

Analyze the shifts and remove conjugate pairs. Add helper array to store the deleted indices.

Solution:

$\rightarrow$ Add a preprocessing to the parallelized loop.

$\rightarrow$ Or use a thread pool (build with PThreads,...).

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Parallelization and Efficiency Improving

Although we only store $\mathcal{P}(L)$ and $\mathcal{P}(U)$ once memory can run short for the value vectors of L_i and U_i.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Parallelization and Efficiency Improving

Although we only store $\mathcal{P}(L)$ and $\mathcal{P}(U)$ once memory can run short for the value vectors of L_i and U_i.

Solution 1: Estimate the maximal number of threads, that can run without swapping the memory:

$$\#threads = \max\left\{ \left\lfloor \frac{s \cdot (M - \text{sizeof}(\mathcal{P}(L)) - \text{sizeof}(\mathcal{P}(U)))}{\text{sizeof}(datatype) \cdot (\text{nnz}(L) + \text{nnz}(U))} \right\rfloor, 1 \right\},$$

where M is the size of the physical memory and s is a security factor.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Parallelization and Efficiency Improving

Although we only store $\mathcal{P}(L)$ and $\mathcal{P}(U)$ once memory can run short for the value vectors of L_i and U_i.

Solution 1: Estimate the maximal number of threads, that can run without swapping the memory:

$$\#threads = \max\left\{ \left\lfloor \frac{s \cdot (M - \text{sizeof}(\mathcal{P}(L)) - \text{sizeof}(\mathcal{P}(U)))}{\text{sizeof}(datatype) \cdot (\text{nnz}(L) + \text{nnz}(U))} \right\rfloor, 1 \right\},$$

where M is the size of the physical memory and s is a security factor.

Solution 2: Use cache-to-disk explicitly. After a decomposition is computed, the values of L and U are written to disk and the memory is freed.
$\rightarrow$ Load them from disk when they are required.
$\rightarrow$ Technique can be used at other points, e.g. in LRCF-ADI, too.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Getting the Initial LU Decomposition

Open Problem: How to compute the initial LU decomposition of A.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Getting the Initial LU Decomposition

Open Problem: How to compute the initial LU decomposition of A.

Use already known software packages:

- CSparse
  - Concise software package.
  - Based on a Left-Looking LU decomposition.
  - Does not neglected numerically zero entries.
  - Easy access to $\mathcal{P}(L)$ and $\mathcal{P}(U)$.
  - Reordering and pivoting controllable.

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Getting the Initial LU Decomposition

Open Problem: How to compute the initial LU decomposition of A.

Use already known software packages:

- CSparse
- UMFPack
  - Used in MATLAB®.
  - Fast, uses Level 3 BLAS.
  - Good access to $\mathcal{P}(L)$ and $\mathcal{P}(U)$.
  - Reordering and pivoting not completly controllable.
  - Seems to be not completely thread-safe.
  - **Neglects numerically zero entries.** $\rightarrow$ **$\mathcal{P}(L)$ and $\mathcal{P}(U)$ sometimes not usable.**

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Getting the Initial LU Decomposition

Open Problem: How to compute the initial LU decomposition of A.

Use already known software packages:

- CSparse
- UMFPack
- SuperLU(-MT)
  - Too fault-sensitive for our application.
  - Access to data structure a bit complicated.
    $\rightarrow$ reject using this for our algorithm

Single-Pattern-Multi-Value Idea for $(A + p_i I)$

Getting the Initial LU Decomposition

Open Problem: How to compute the initial LU decomposition of A.

Use already known software packages:

- CSparse
- UMFPack
- SuperLU(-MT)

$\rightarrow$ In general every algorithm which provide $\mathcal{P}_r(L, *)$ and $\mathcal{P}_r(U, *)$ can be used.

Numerical Results

Numerical Results

SPMV LU decomposition - Memory Usage

Memory usage for $A + p_i I$ (and $(A + p_i E)$) with $r = 16$ on a 64bit machine:

	N	L+U in MB	16 LUs in MB	SPMV[1] LU	savings
FDM	10 000	6.45	103.20	53.68	47.99%
FDM	90 000	90.75	1 452.00	760.91	47.60%
FDM	250 000	285.10	4 561.30	2 394.50	47.50%
FDM	562 500	718.00	11 488.00	6 038.00	47.44%
FDM	1 000 000	1 379.00	22 064.00	11 604.00	47.41%
Rail	79 841	54.11	865.76	450.80	47.93%
Filter3D	106 437	888.05	14 208.80	7 536.22	46.96%

[1] single-pattern-multi-value

Numerical Results

SPMV LU decomposition - Memory Usage

Memory usage for $A + p_i I$ (and $(A + p_i E)$) with $r = 16$ on a 64bit machine:

	N	L+U in MB	16 LUs in MB	SPMV[1] LU	savings
FDM	10 000	6.45	103.20	53.68	47.99%
FDM	90 000	90.75	1 452.00	760.91	47.60%
FDM	250 000	285.10	4 561.30	2 394.50	47.50%
FDM	562 500	718.00	11 488.00	6 038.00	47.44%
FDM	1 000 000	1 379.00	22 064.00	11 604.00	47.41%
Rail	79 841	54.11	865.76	450.80	47.93%
Filter3D	106 437	888.05	14 208.80	7 536.22	46.96%

$\rightarrow$ Reduce memory usage drastically.

[1] single-pattern-multi-value

Numerical Results

SPMV LU decomposition - Memory Usage

Memory usage for $A + p_i I$ (and $(A + p_i E)$) with $r = 16$ on a 64bit machine:

	N	L+U in MB	16 LUs in MB	SPMV[1] LU	savings
FDM	10 000	6.45	103.20	53.68	47.99%
FDM	90 000	90.75	1 452.00	760.91	47.60%
FDM	250 000	285.10	4 561.30	2 394.50	47.50%
FDM	562 500	718.00	11 488.00	6 038.00	47.44%
FDM	1 000 000	1 379.00	22 064.00	11 604.00	47.41%
Rail	79 841	54.11	865.76	450.80	47.93%
Filter3D	106 437	888.05	14 208.80	7 536.22	46.96%

$\rightarrow$ Reduce memory usage drastically.

$\rightarrow$ Using cache-to-disk only one LU decomposition is held in memory.

[1] single-pattern-multi-value

Numerical Results

SPMV LU decomposition - Pattern Reuse

Computing a LU decomposition on a quad 8-core Intel® Xeon® E7-8837, 2.67GHz with 1TB RAM:

	Problem	LU with CSparse	LU with known $\mathcal{P}(L)$, $\mathcal{P}(U)$	savings
FDM	10 000	0.135	0.056	59.00%
FDM	90 000	1.621	0.781	51.80%
FDM	250 000	6.905	3.461	49.88%
FDM	562500	28.838	21.756	24.56%
FDM	1 000 000	88.219	67.967	22.96%
Rail	79 841	0.583	0.252	56.80%
Filter3D	106 437	104.165	91.867	11.81%

Numerical Results

SPMV LU decomposition - Pattern Reuse

Computing a LU decomposition on a quad 8-core Intel® Xeon® E7-8837, 2.67GHz with 1TB RAM:

	Problem	LU with CSparse	LU with known $\mathcal{P}(L)$, $\mathcal{P}(U)$	savings
FDM	10 000	0.135	0.056	59.00%
FDM	90 000	1.621	0.781	51.80%
FDM	250 000	6.905	3.461	49.88%
FDM	562500	28.838	21.756	24.56%
FDM	1 000 000	88.219	67.967	22.96%
Rail	79 841	0.583	0.252	56.80%
Filter3D	106 437	104.165	91.867	11.81%

$\rightarrow$ The reuse technique can fasten the decomposition.

Numerical Results

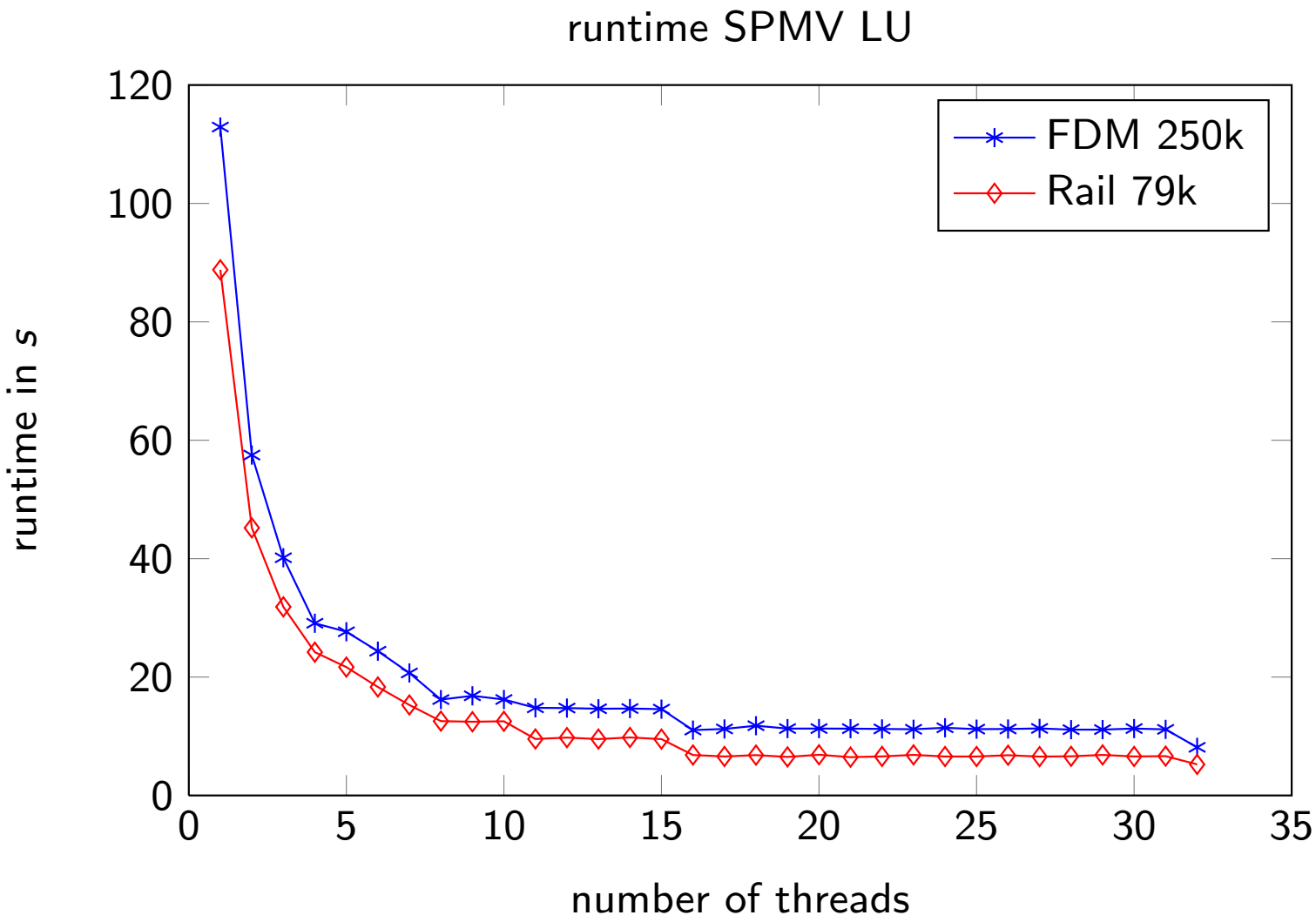
SPMV LU decomposition - Parallelization

We use a set of 32 parameters on the same machine:

	FDM 250 000		Rail 79 841	
#Threads	Time	Speedup	Time	Speedup
1	112.90	1.00	88.79	1.00
2	57.50	1.96	45.19	1.96
4	29.09	3.88	24.20	3.67
8	16.20	6.97	12.55	7.07
16	11.04	10.23	6.83	13.00
32	8.13	13.88	5.25	16.92

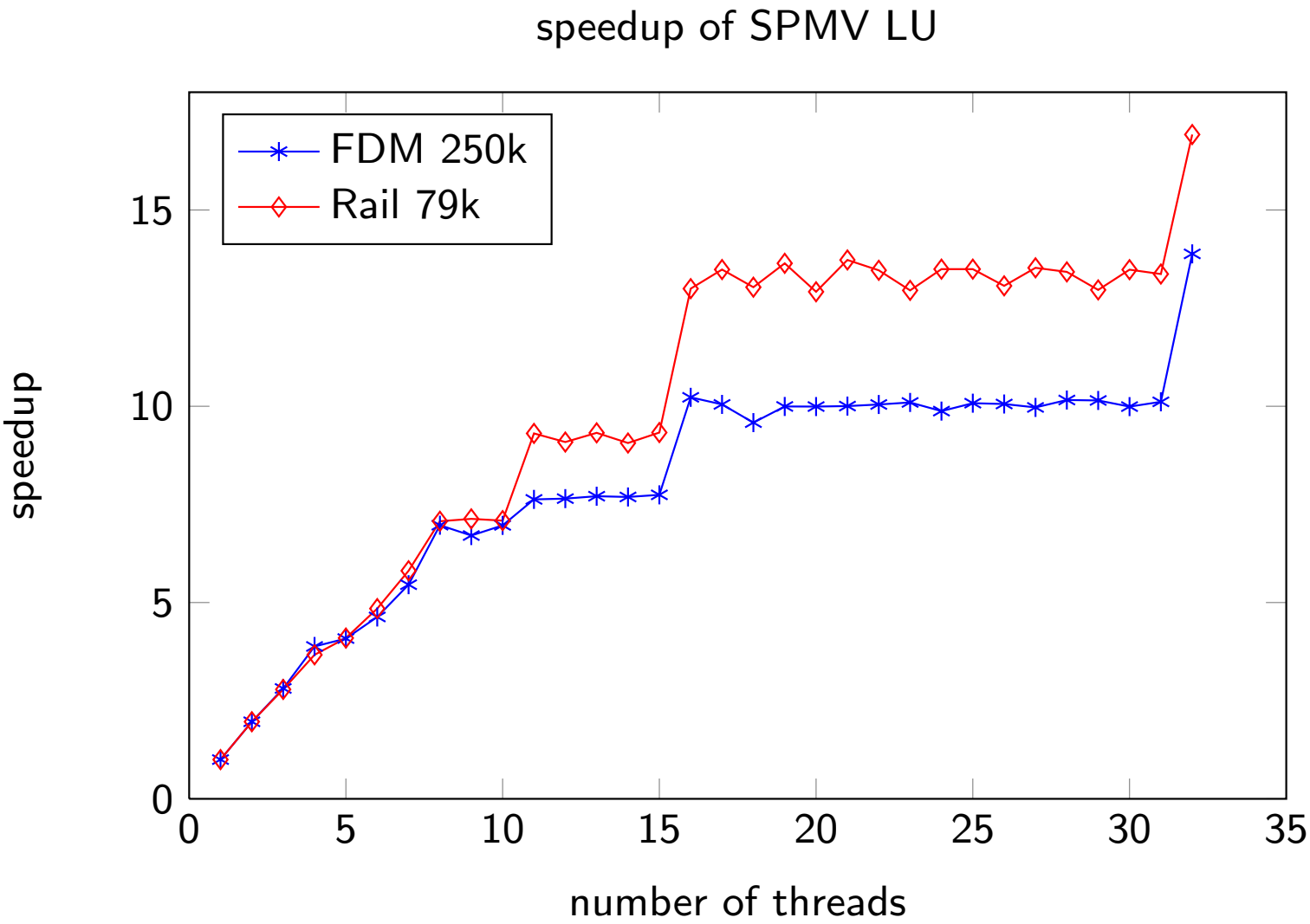
Numerical Results

SPMV LU decomposition - Parallelization



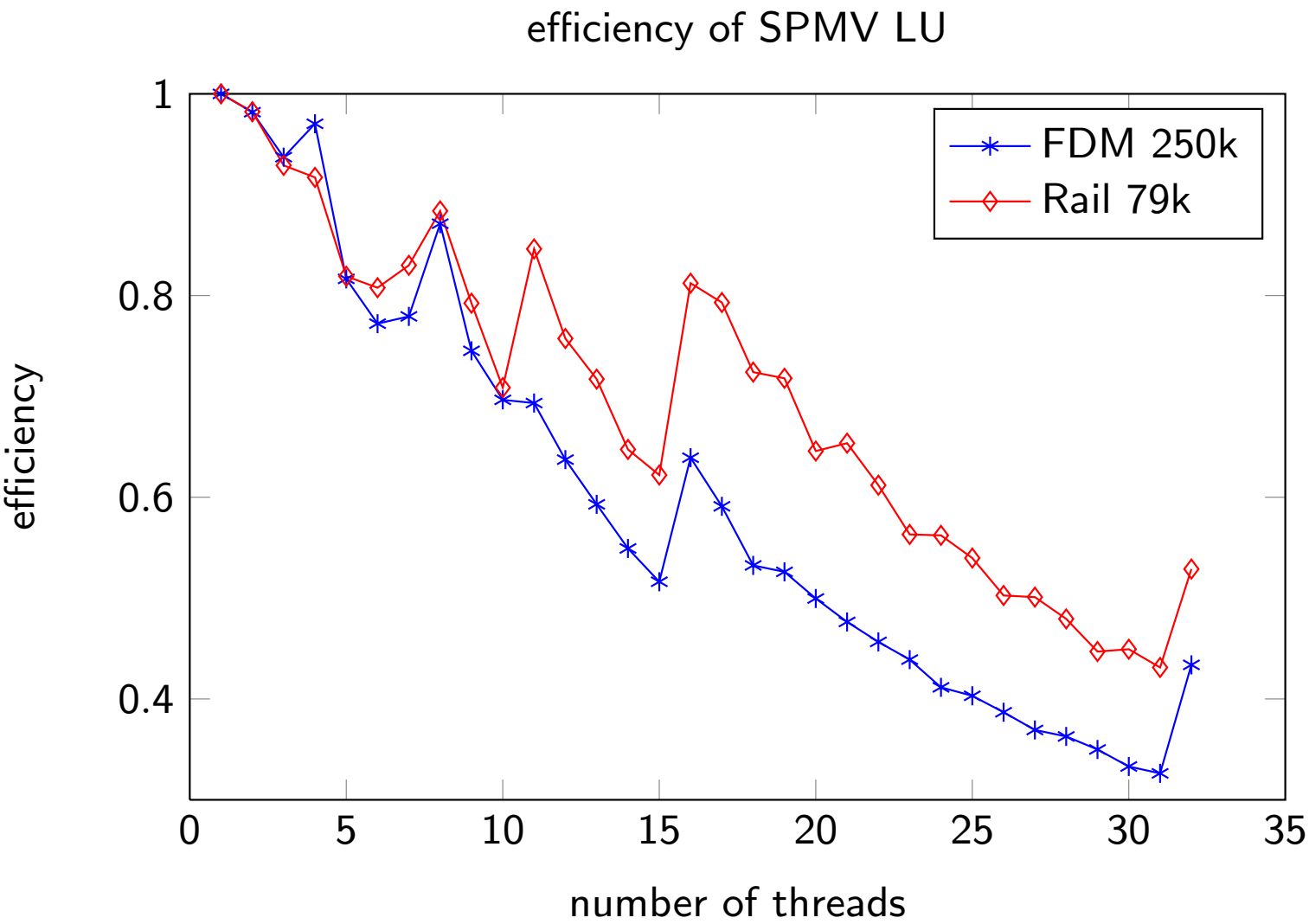
Numerical Results

SPMV LU decomposition - Parallelization



Numerical Results

SPMV LU decomposition - Parallelization



Numerical Results

SPMV LU decomposition - Parallelization

We use a set of 32 parameters on the same machine:

	FDM 250 000		Rail 79 841	
#Threads	Time	Speedup	Time	Speedup
1	112.90	1.00	88.79	1.00
2	57.50	1.96	45.19	1.96
4	29.09	3.88	24.20	3.67
8	16.20	6.97	12.55	7.07
16	11.04	10.23	6.83	13.00
32	8.13	13.88	5.25	16.92

$\rightarrow$ The parallelization save much time.
$\rightarrow$ Efficiency is bounded by memory bandwidth.

Numerical Results

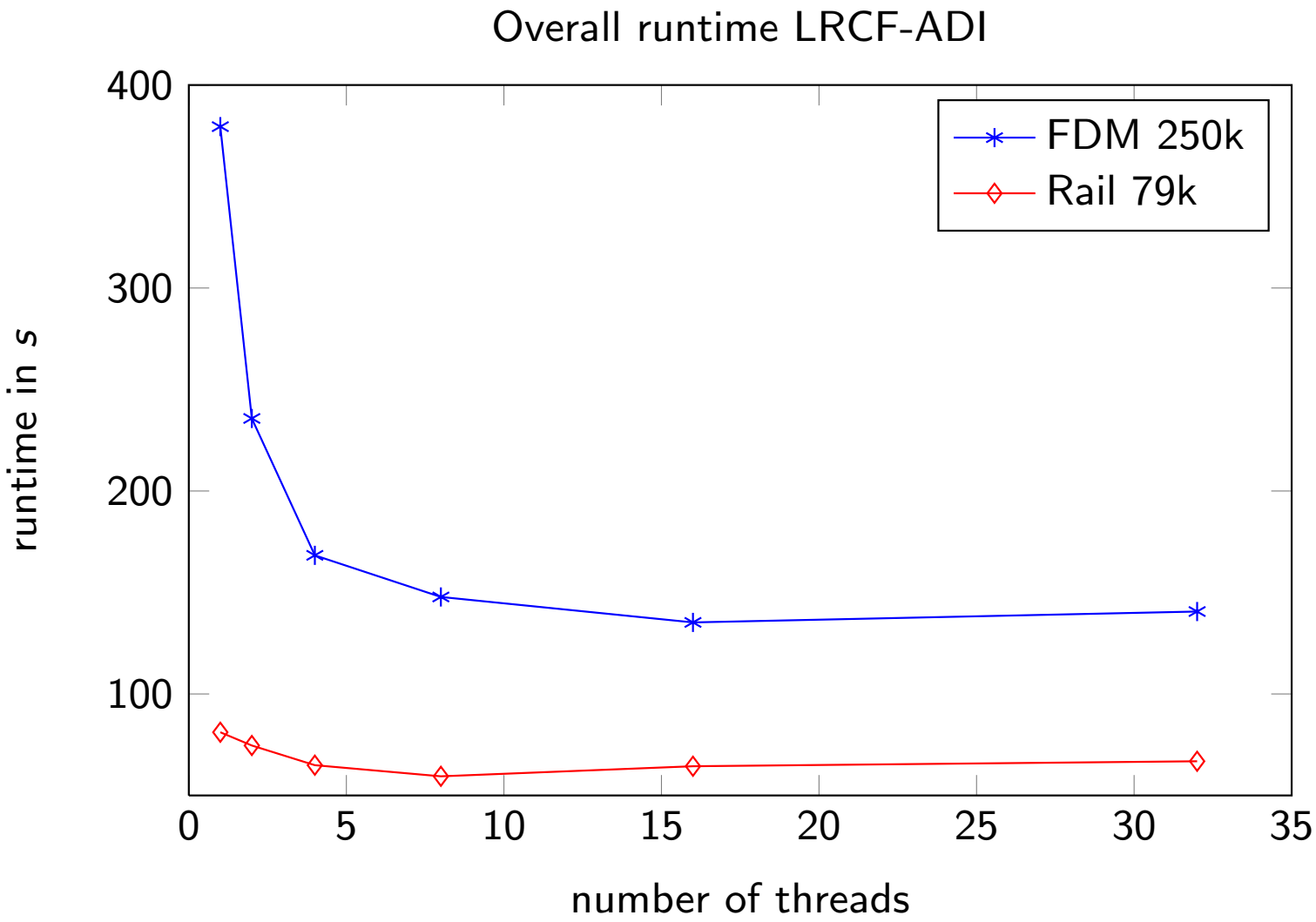
Overall Process: LRCF-ADI

Solving a (generalized) Lyapunov Equation using LRCF-ADI with 16 shift parameters:

	FDM 250 000		Rail 79 841	
#Threads	Time	Speedup	Time	Speedup
1	379.49	1.00	81.14	1.00
2	235.74	1.61	74.61	1.09
4	168.30	2.25	64.91	1.25
8	147.79	2.57	59.44	1.37
16	135.25	2.81	64.39	1.26
32	140.61	2.70	66.85	1.21

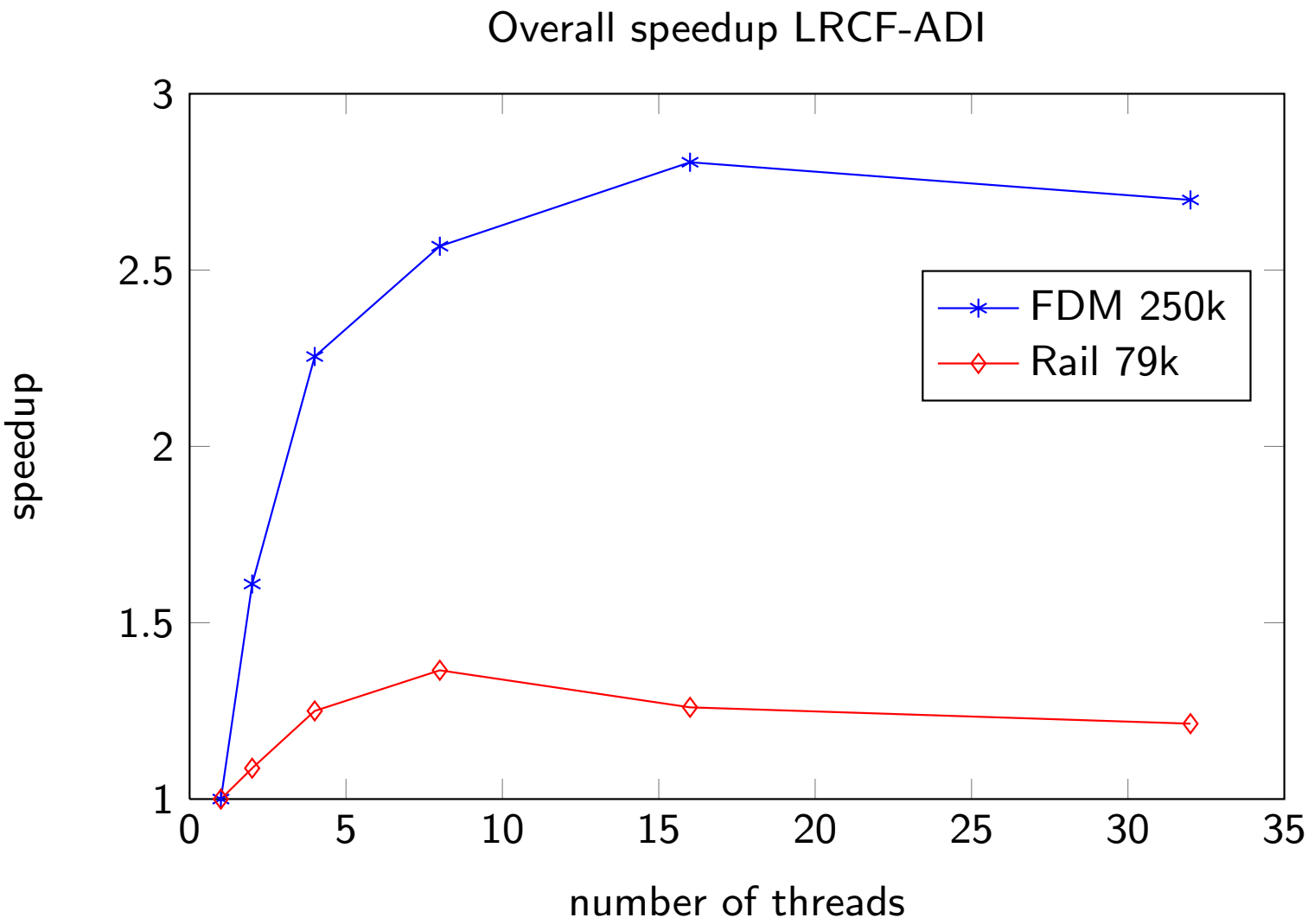
Numerical Results

Overall Process: LRCF-ADI



Numerical Results

Overall Process: LRCF-ADI



Numerical Results

Overall Process: LRCF-ADI

Solving a (generalized) Lyapunov Equation using LRCF-ADI with 16 shift parameters:

	FDM 250 000		Rail 79 841	
#Threads	Time	Speedup	Time	Speedup
1	379.49	1.00	81.14	1.00
2	235.74	1.61	74.61	1.09
4	168.30	2.25	64.91	1.25
8	147.79	2.57	59.44	1.37
16	135.25	2.81	64.39	1.26
32	140.61	2.70	66.85	1.21

$\rightarrow$ Too much non parallelizable code.
$\rightarrow$ Memory bandwidth problem is getting bigger.
$\rightarrow$ Too many transfers between different physical CPUs.

Numerical Results

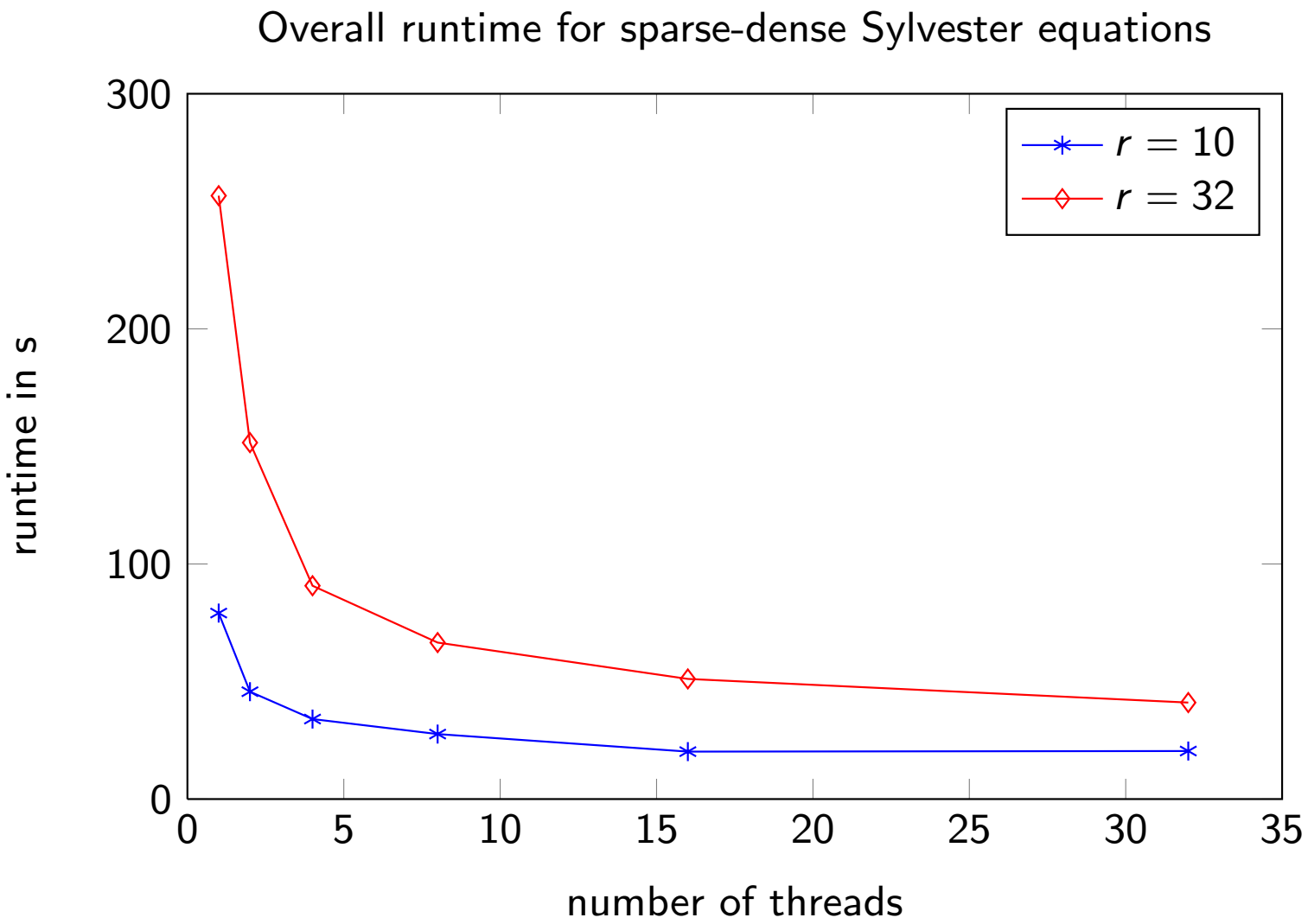
Overall Process: Sylvester Solver

Solve a standard sparse-dense Sylvester equation with the 250 000 FDM matrix and different small matrices:

	$r = 10$		$r = 32$	
#Threads	Time	Speedup	Time	Speedup
1	79.11	1.00	256.65	1.00
2	45.67	1.73	151.61	1.69
4	34.03	2.32	90.74	2.83
8	27.65	2.86	66.57	3.86
16	20.19	3.92	51.10	5.02
32	20.43	3.87	41.05	6.25

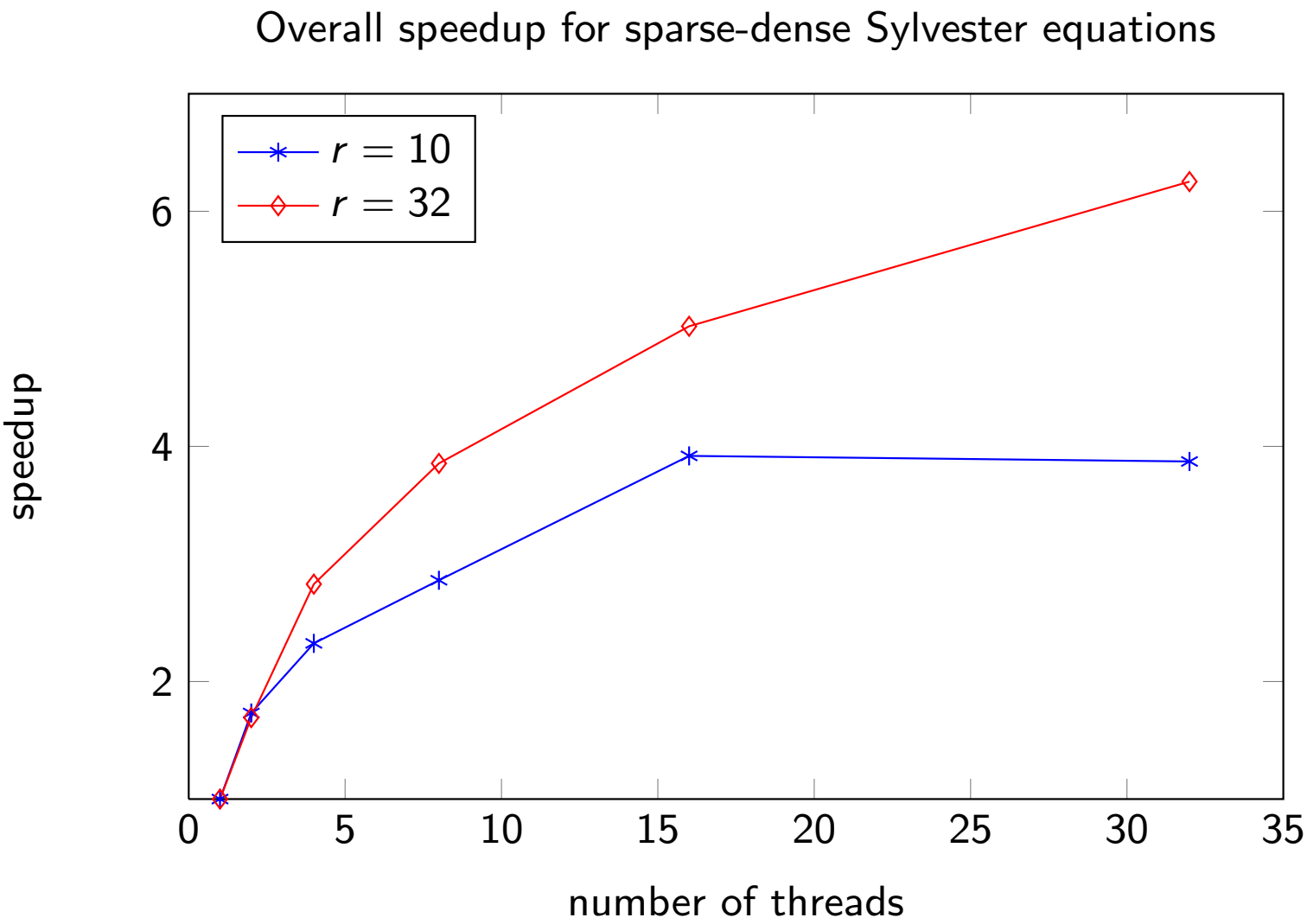
Numerical Results

Overall Process: Sylvester Solver



Numerical Results

Overall Process: Sylvester Solver



Numerical Results

Overall Process: Sylvester Solver

Solve a standard sparse-dense Sylvester equation with the 250 000 FDM matrix and different small matrices:

	$r = 10$		$r = 32$	
#Threads	Time	Speedup	Time	Speedup
1	79.11	1.00	256.65	1.00
2	45.67	1.73	151.61	1.69
4	34.03	2.32	90.74	2.83
8	27.65	2.86	66.57	3.86
16	20.19	3.92	51.10	5.02
32	20.43	3.87	41.05	6.25

$\rightarrow$ Ratio between data size and number of threads is important.

Conclusion and Outlook

Conclusions

- The memory usage can be reduced drastically.
- Memory can be preallocated in large blocks.
- The reuse technique can accelerate the factorizations significantly.
- Even systems up to a dimension of $> 250\,000$ with more than 15 parameters can be solved on current workstations.
- Memory bandwidth limits the speedup more than other factors.
- OpenMP is a good tool to parallelize code, but is not the holy grail. A combination with PThreads is useful.

Conclusion and Outlook

Open Problems

- Ideas porting the reuse idea to left-looking and multi-frontal LU.
- Robust parallel generation of $\mathcal{P}(L)$ and $\mathcal{P}(U)$.
- Algorithms that reduce the required memory bandwidth.
- Ideas for direct sparse solver on GPUs.
- Other parallelization ideas for this algorithm(s).
- Pivoting can destroy the factorization.

Conclusion and Outlook

Open Problems

- Ideas porting the reuse idea to left-looking and multi-frontal LU.
- Robust parallel generation of $\mathcal{P}(L)$ and $\mathcal{P}(U)$.
- Algorithms that reduce the required memory bandwidth.
- Ideas for direct sparse solver on GPUs.
- Other parallelization ideas for the algorithm(s).
- Pivoting can destroy the factorization.

That's it.
Thanks for your attention.