A NEW PARALLEL SOLVER FOR SPARSE SYMMETRIC MATRICES (SYMPACK) AND A STORAGE OPTIMAL SELECTED INVERSION (PEXSI)

Mathias Jacquelin
mjacquelin@lbl.gov

Weile Jia, Lin Lin, Chao Yang
Esmond Ng, Yili Zheng, Kathy Yelick

August 16 2017

Scalable Solvers Group
Computational Research Department
Lawrence Berkeley National Laboratory

OUTLINE

OUTLINE

Motivations:

- Sparse matrices arise in many applications:
  - Optimization problems
  - Discretized PDEs
  - Electronic structure theory
  - ...
- Some sparse direct methods require:
  - Sparse factorizations
  - Computing some inverse elements

Motivations:

- Sparse matrices arise in many applications:
  - Optimization problems
  - Discretized PDEs
  - Electronic structure theory
  - ...
- Some sparse direct methods require:
  - Sparse factorizations
  - **Computing some inverse elements**

Motivations:

- Sparse matrices arise in many applications:
  - Optimization problems
  - Discretized PDEs
  - Electronic structure theory
  - ...
- Some sparse direct methods require:
  - Sparse factorizations
  - **Computing some inverse elements**

Challenges for current and future platforms:

- Lower amount of memory per core
- Higher relative communication costs
- Large performance variations

OUTLINE

Motivation:

- Many applications require **some** elements of an inverse matrix:
  - Electronic structure theory
  - Confidence interval estimation
  - Poisson-Boltzmann
  - Quantum transport theory

Ex: Kohn-Sham density functional theory

$$\Gamma = f(H - \mu I) \approx \sum_{i=1}^{Q} \frac{\omega_i}{H - z_i I}$$

$$\rho(\Gamma) \approx \sum_{i=1}^{Q} diag\left(\frac{\omega_i}{H - z_i I}\right)$$

Objective:

- Compute **selected entries** of A^{-1}

$$\{(A^{-1})_{ij} \mid A_{ij} \neq 0 \ \ or \ \ A_{ji} \neq 0, 1 \leq i,j \leq N\}$$

SELECTED INVERSION

- "Naive" solution: sequence of solve on e_j
  - $A = LU, A^{-1} = [x_1, x_2 \dots x_N] \quad \Longrightarrow \quad$ Solve $LUx_j = e_j$
- Better algorithms:

  [Takahashi et al. 1973], [Erismann and Tinney 1975]

SELECTED INVERSION

- "Naive" solution: sequence of solve on e_j
  - $A = LU, A^{-1} = [x_1, x_2 \dots x_N] \implies$ Solve $LUx_j = e_j$
- Better algorithms:

  [Takahashi et al. 1973], [Erismann and Tinney 1975]

- Let $A = LU$ be the LU factorization of A (or $A = LDL^T$ if A is symmetric)
- Selected Inversion: entries of A^{-1} corresponding to $\Omega(A)$

SELECTED INVERSION

- "Naive" solution: sequence of solve on e_j
  - $A = LU, A^{-1} = [x_1, x_2 \dots x_N] \implies$ Solve $LUx_j = e_j$
- Better algorithms:

  [Takahashi et al. 1973], [Erismann and Tinney 1975]

- Let $A = LU$ be the LU factorization of A (or $A = LDL^T$ if A is symmetric)
- Selected Inversion: entries of A^{-1} corresponding to $\Omega(A)$
- If A sparse, entries of A^{-1} in $\Omega(A)$ require entries in $\Omega(L + U)$

SELECTED INVERSION

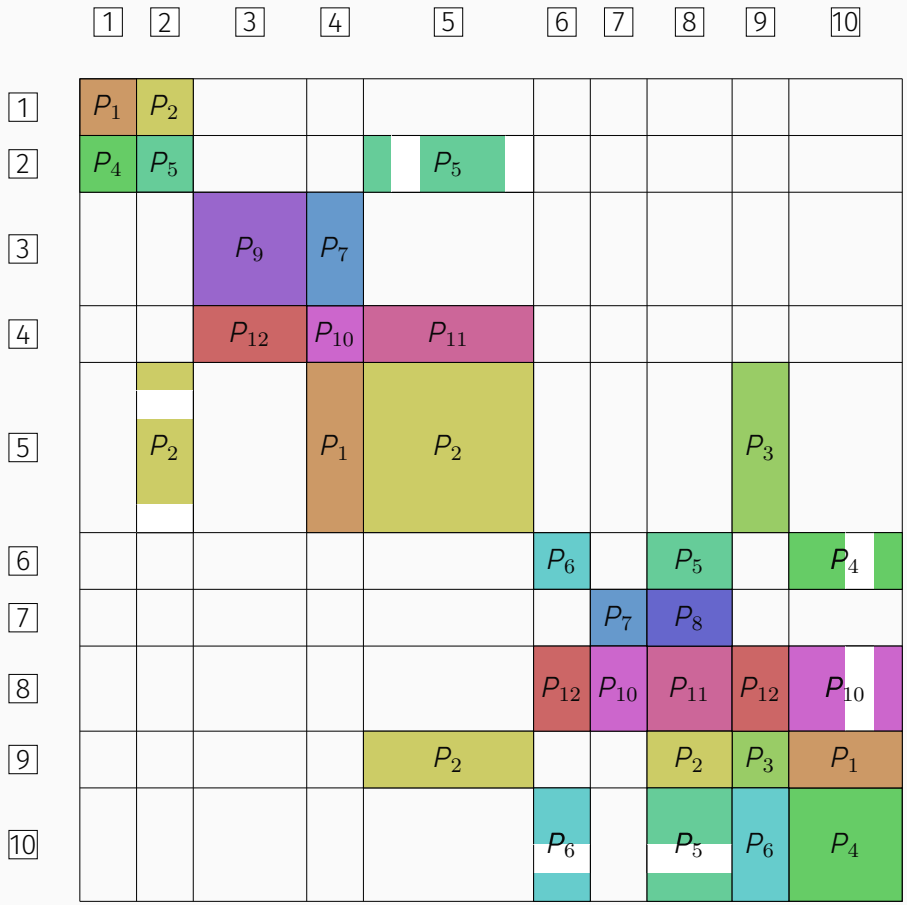
- "Naive" solution: sequence of solve on e_j
  - $A = LU, A^{-1} = [x_1, x_2 \dots x_N] \quad \Longrightarrow \quad$ Solve $LUx_j = e_j$
- Better algorithms:

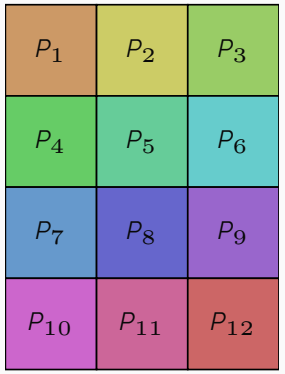
  [Takahashi et al. 1973], [Erismann and Tinney 1975]

- Let $A = LU$ be the LU factorization of A (or $A = LDL^T$ if A is symmetric)
- Selected Inversion: entries of A^{-1} corresponding to $\Omega(A)$
- If A sparse, entries of A^{-1} in $\Omega(A)$ require entries in $\Omega(L + U)$
- All entries of A^{-1} in $\Omega(L + U)$ are actually computed

SPARSE MATRIX 2D BLOCK LAYOUT

for *Supernode* $\mathcal{K} = \mathcal{N}$ *down to* 1 **do**
  Compute selected elements of $A^{-}1$ within $\mathcal{K}$
end



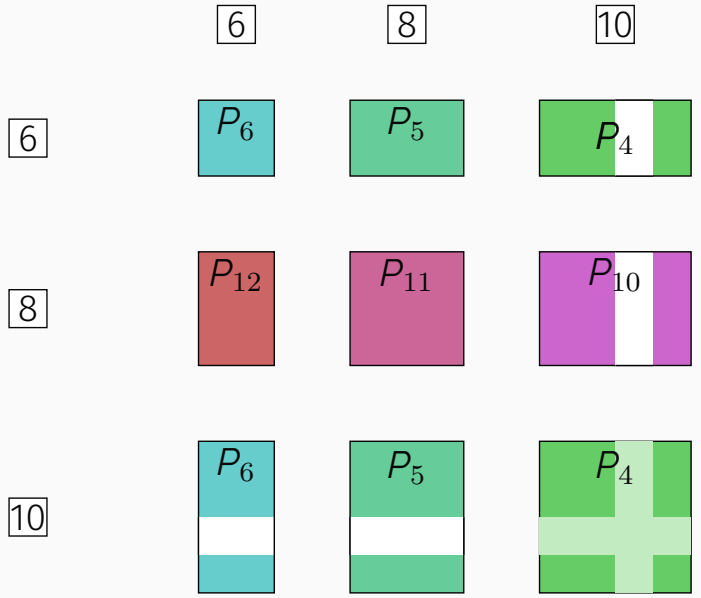


- 2D Block Cyclic layout
- 4-by-3 processor grid
- No explicit load balancing
- Works well in practice [Gupta]

PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + U\ S_{-1}L & -US^{-1} \\ -S^{-1}L & S^{-1} \end{pmatrix}$$

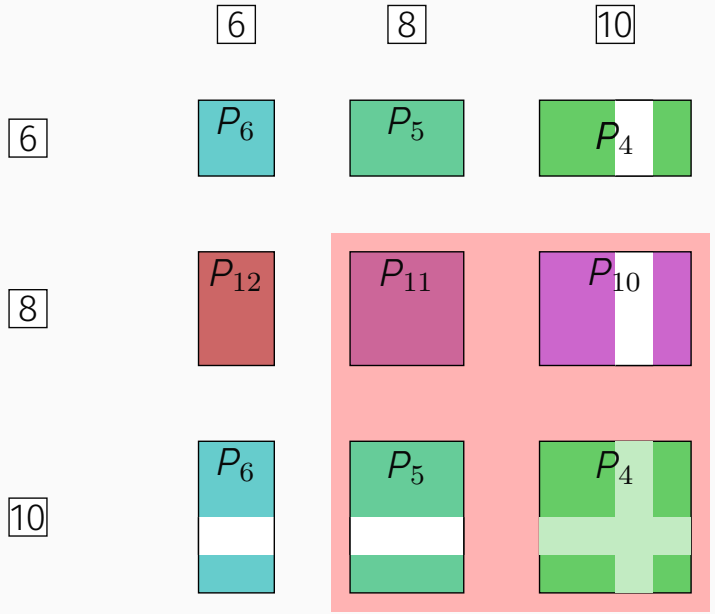
- Supernode depends on S^{-1}



PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + U\ S_{-1}L & -US^{-1} \\ -S^{-1}L & S^{-1} \end{pmatrix}$$

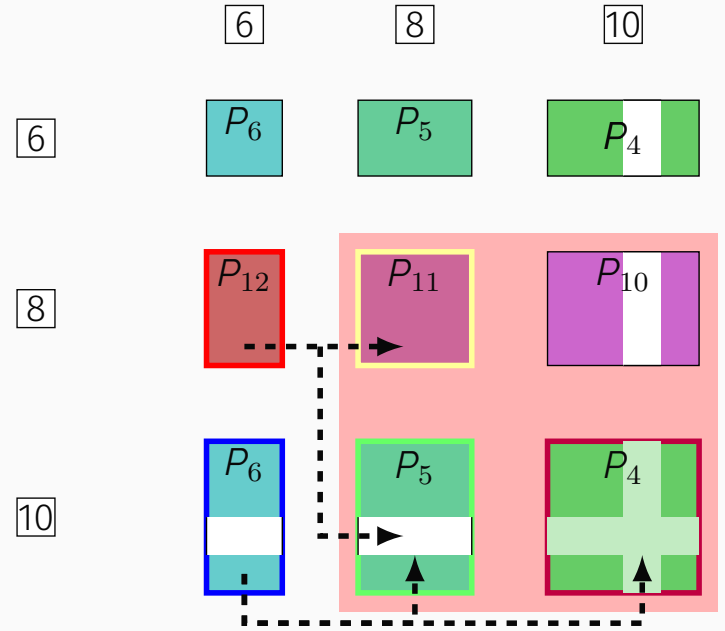
- Supernode depends on S^{-1}
- Supernode processed in parallel
  - $S^{-1} \leftrightarrow$ Ancestors in the elimination tree
  - Ancestors compute contributions
  - Contributions reduced



PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + U\ S_{-1}L & -US^{-1} \\ -S^{-1}L & S^{-1} \end{pmatrix}$$

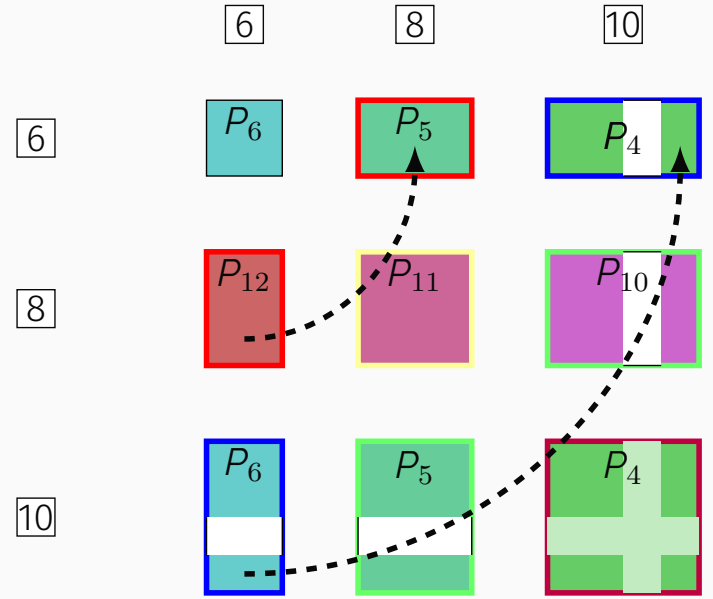
- Supernode depends on S^{-1}
- Supernode processed in parallel
  - $S^{-1} \leftrightarrow$ Ancestors in the elimination tree
  - Ancestors compute contributions
  - Contributions reduced
- Complex communication pattern



PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + U\ S_{-1}L & -US^{-1} \\ -S^{-1}L & S^{-1} \end{pmatrix}$$

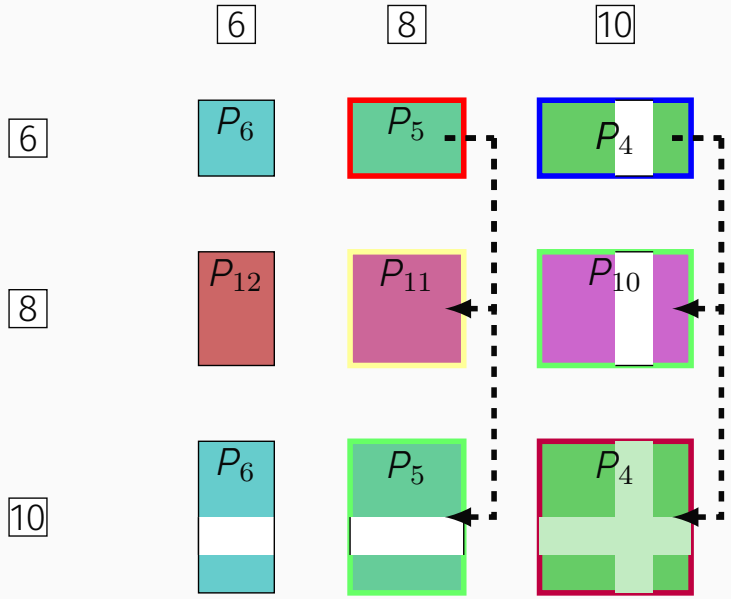
- Supernode depends on S^{-1}
- Supernode processed in parallel
  - $S^{-1} \leftrightarrow$ Ancestors in the elimination tree
  - Ancestors compute contributions
  - Contributions reduced
- Complex communication pattern
  - Sending to cross-diagonal processors $\leftrightarrow$ simpler pattern



PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + U\ S_{-1}L & -US^{-1} \\ -S^{-1}L & S^{-1} \end{pmatrix}$$

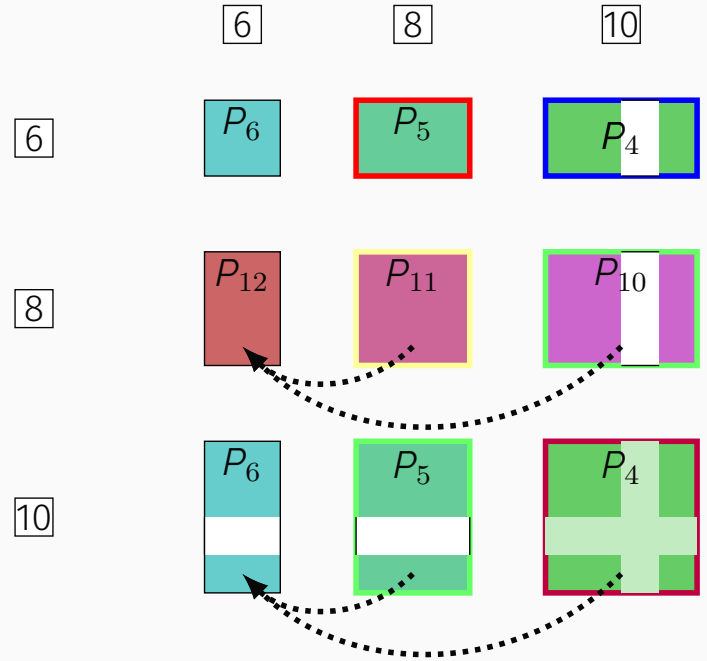
- Supernode depends on S^{-1}
- Supernode processed in parallel
  - $S^{-1} \leftrightarrow$ Ancestors in the elimination tree
  - Ancestors compute contributions
  - Contributions reduced
- Complex communication pattern
  - Sending to cross-diagonal processors $\leftrightarrow$ simpler pattern
  - Communication only within rows / columns of processors



PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + U\ S_{-1}L & -US^{-1} \\ -S^{-1}L & S^{-1} \end{pmatrix}$$

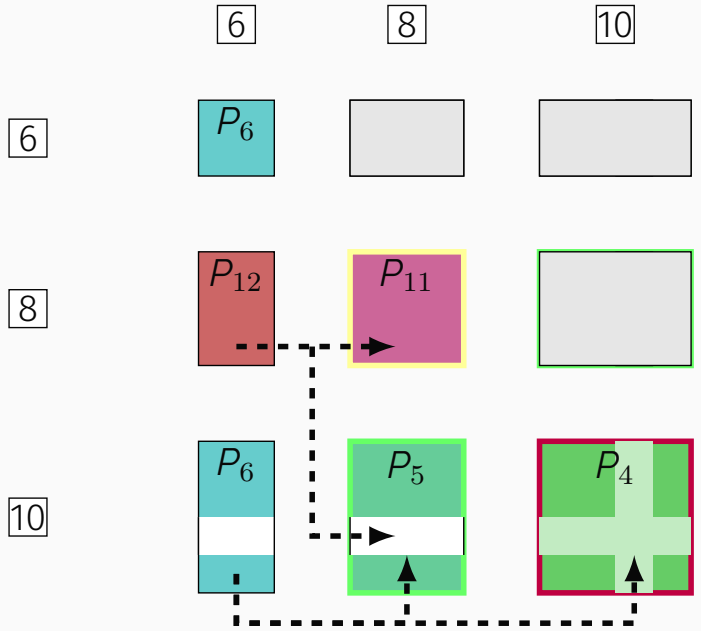
- Supernode depends on S^{-1}
- Supernode processed in parallel
  - $S^{-1} \leftrightarrow$ Ancestors in the elimination tree
  - Ancestors compute contributions
  - Contributions reduced
- Complex communication pattern
  - Sending to cross-diagonal processors $\leftrightarrow$ simpler pattern
  - Communication only within rows / columns of processors



PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$

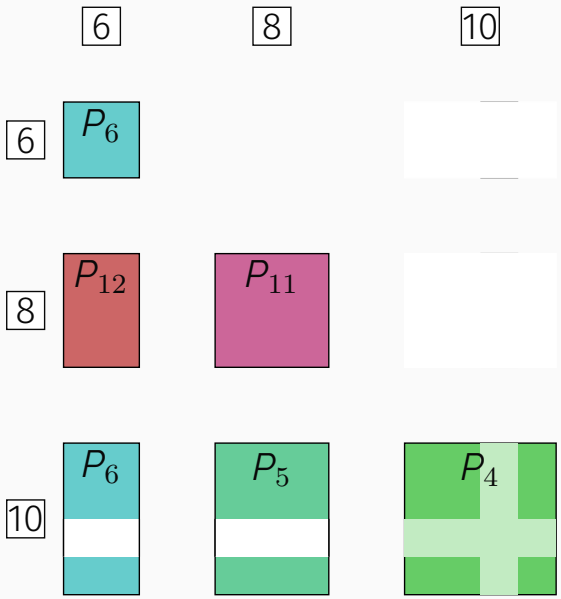
- Supernode depends on S^{-1}
- Supernode processed in parallel
  - $S^{-1} \leftrightarrow$ Ancestors in the elimination tree
  - Ancestors compute contributions
  - Contributions reduced
- Complex communication pattern
  - ~~Sending to cross-diagonal processors $\leftrightarrow$ simpler pattern~~
  - ~~Communication only within rows / columns of processors~~



PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$

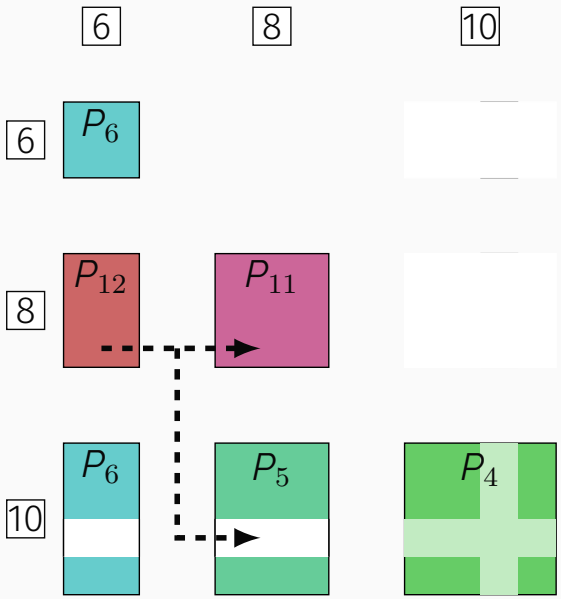
- D: diagonal block
- L: lower triangular block



PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$

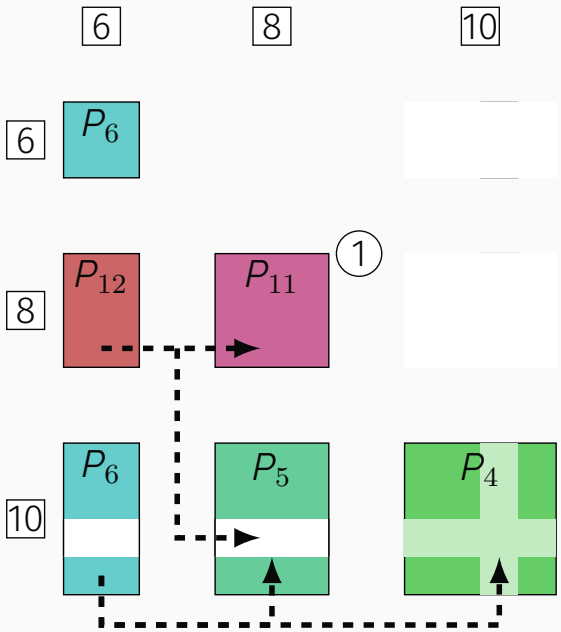
- D: diagonal block
- L: lower triangular block
- Bcast. L to appropriate procs.



PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$

- D: diagonal block
- L: lower triangular block

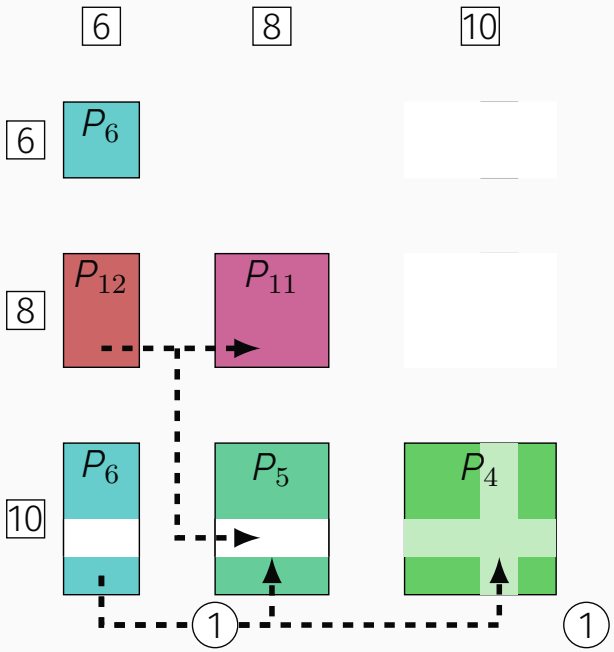


- Bcast. L to appropriate procs.
- Diagonal computes $-S^{-1}L$ only

PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$

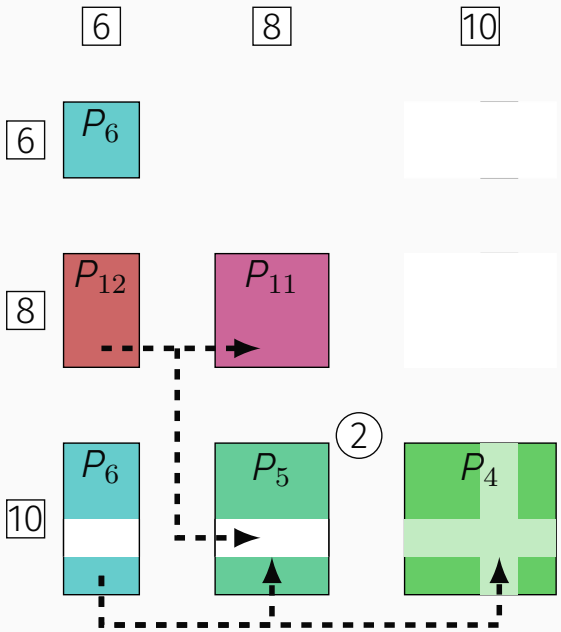
- D: diagonal block
- L: lower triangular block



- Bcast. L to appropriate procs.
- Diagonal computes $-S^{-1}L$ only

PARALLEL PROCESSING OF A SUPERNODE

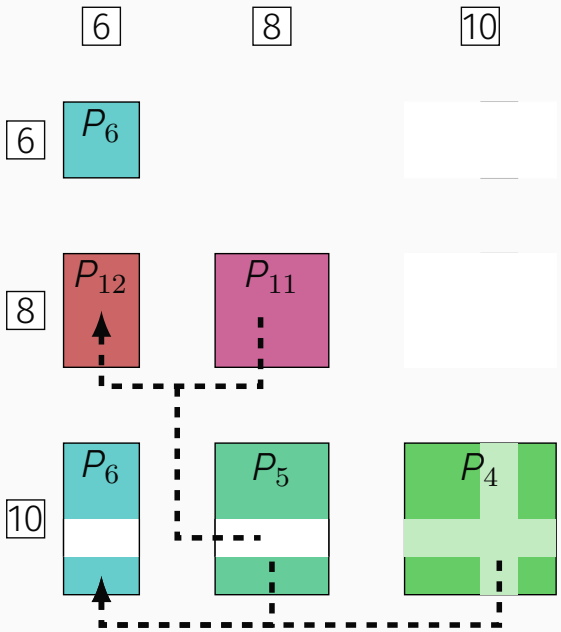
$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$



- D: diagonal block
- L: lower triangular block
- Bcast. L to appropriate procs.
- Diagonal computes $-S^{-1}L$ only
- Off-diagonal also computes $-S^{-T}L$

PARALLEL PROCESSING OF A SUPERNODE

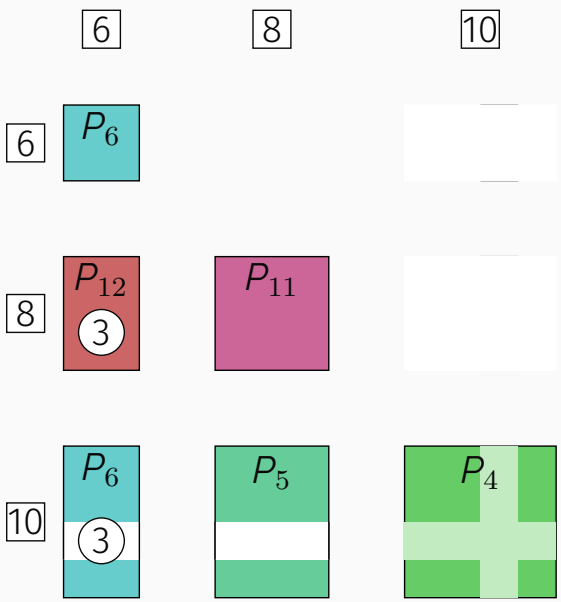
$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$



- D: diagonal block
- L: lower triangular block
- Bcast. L to appropriate procs.
- Diagonal computes $-S^{-1}L$ only
- Off-diagonal also computes $-S^{-T}L$
- Reduce contrib. to L

PARALLEL PROCESSING OF A SUPERNODE

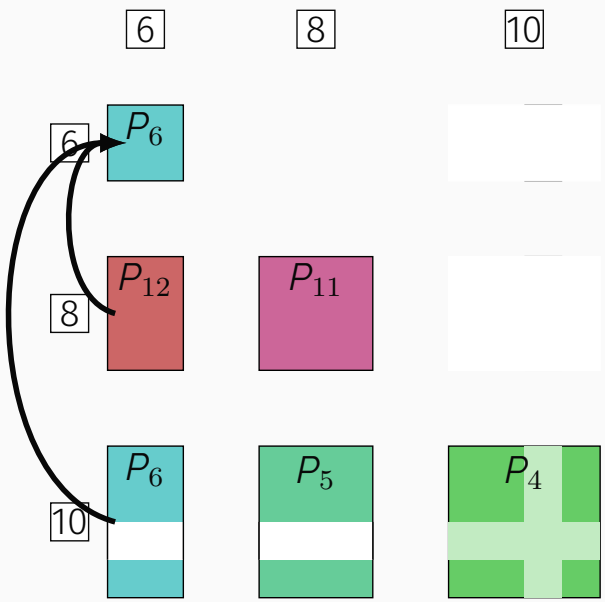
$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$



- D: diagonal block
- L: lower triangular block
- Bcast. L to appropriate procs.
- Diagonal computes $-S^{-1}L$ only
- Off-diagonal also computes $-S^{-T}L$
- Reduce contrib. to L
- Compute contrib. to D

PARALLEL PROCESSING OF A SUPERNODE

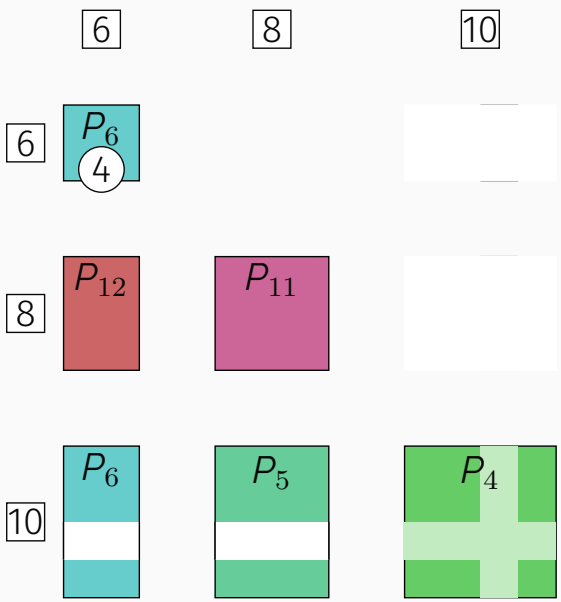
$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$



- D: diagonal block
- L: lower triangular block
- Bcast. L to appropriate procs.
- Diagonal computes $-S^{-1}L$ only
- Off-diagonal also computes $-S^{-T}L$
- Reduce contrib. to L
- Compute contrib. to D
- Reduce contrib. to D

PARALLEL PROCESSING OF A SUPERNODE

$$A^{-1} = \begin{pmatrix} D^{-1} + L^T S_{-1} L & \\ -S^{-1} L & S^{-1} \end{pmatrix}$$



- D: diagonal block
- L: lower triangular block
- Bcast. L to appropriate procs.
- Diagonal computes $-S^{-1}L$ only
- Off-diagonal also computes $-S^{-T}L$
- Reduce contrib. to L
- Compute contrib. to D
- Reduce contrib. to D

PERFORMANCE EVALUATION

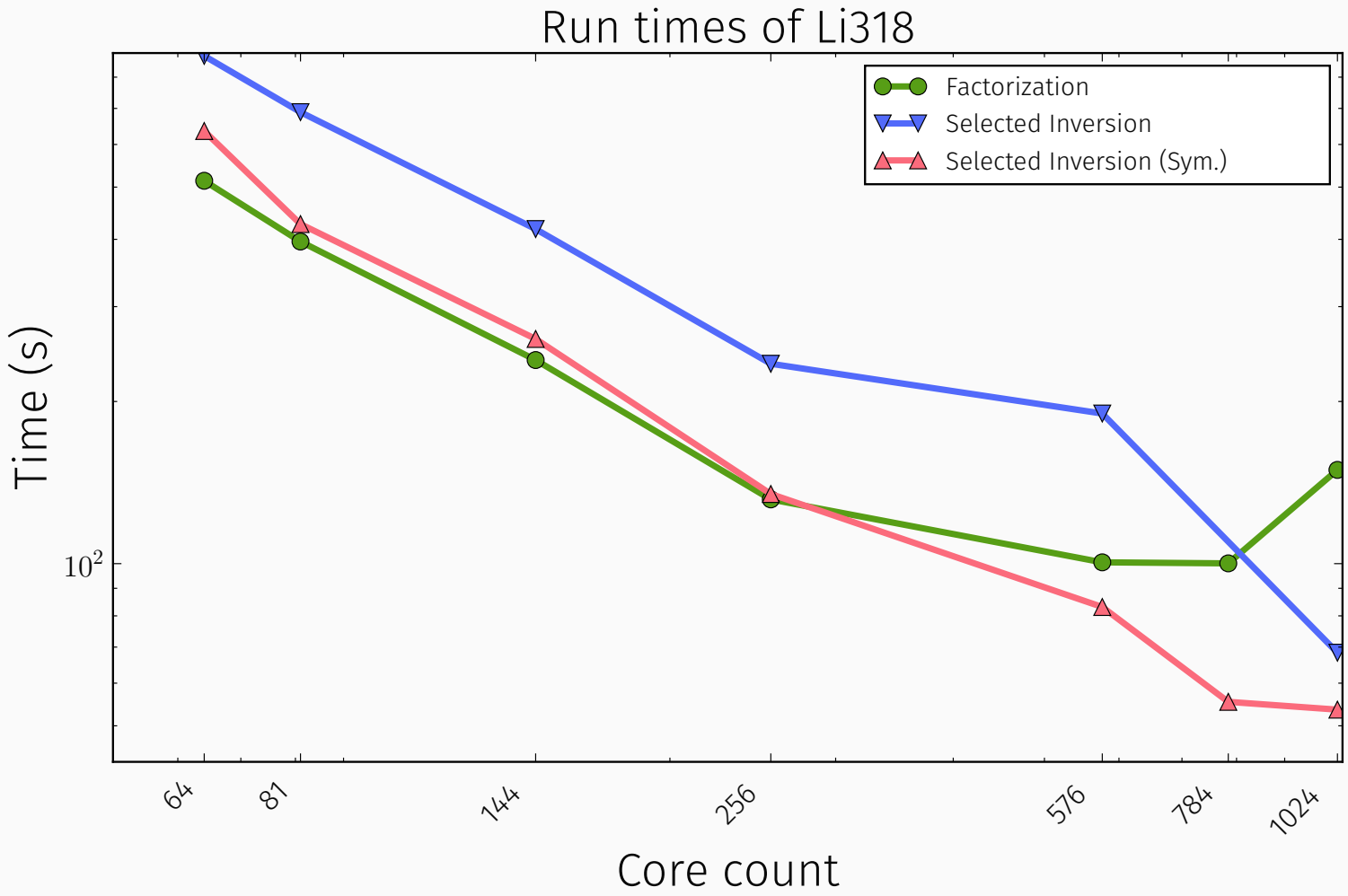
- Experiments on NERSC Cori
  - Intel Haswell processors
  - 32 cores per node (two sockets)
  - 4 GB of memory per core
- `ParMETIS` used for matrix ordering
- `SuperLU_DIST` used for factorization
- `PSelInv`:
  - "Flat" MPI implementation
  - Only asynchronous P2P send/recv

PERFORMANCE EVALUATION

- Experiments on NERSC Cori
  - Intel Haswell processors
  - 32 cores per node (two sockets)
  - 4 GB of memory per core
- `ParMETIS` used for matrix ordering
- `SuperLU_DIST` used for factorization
- `PSelInv`:
  - "Flat" MPI implementation
  - Only asynchronous P2P send/recv

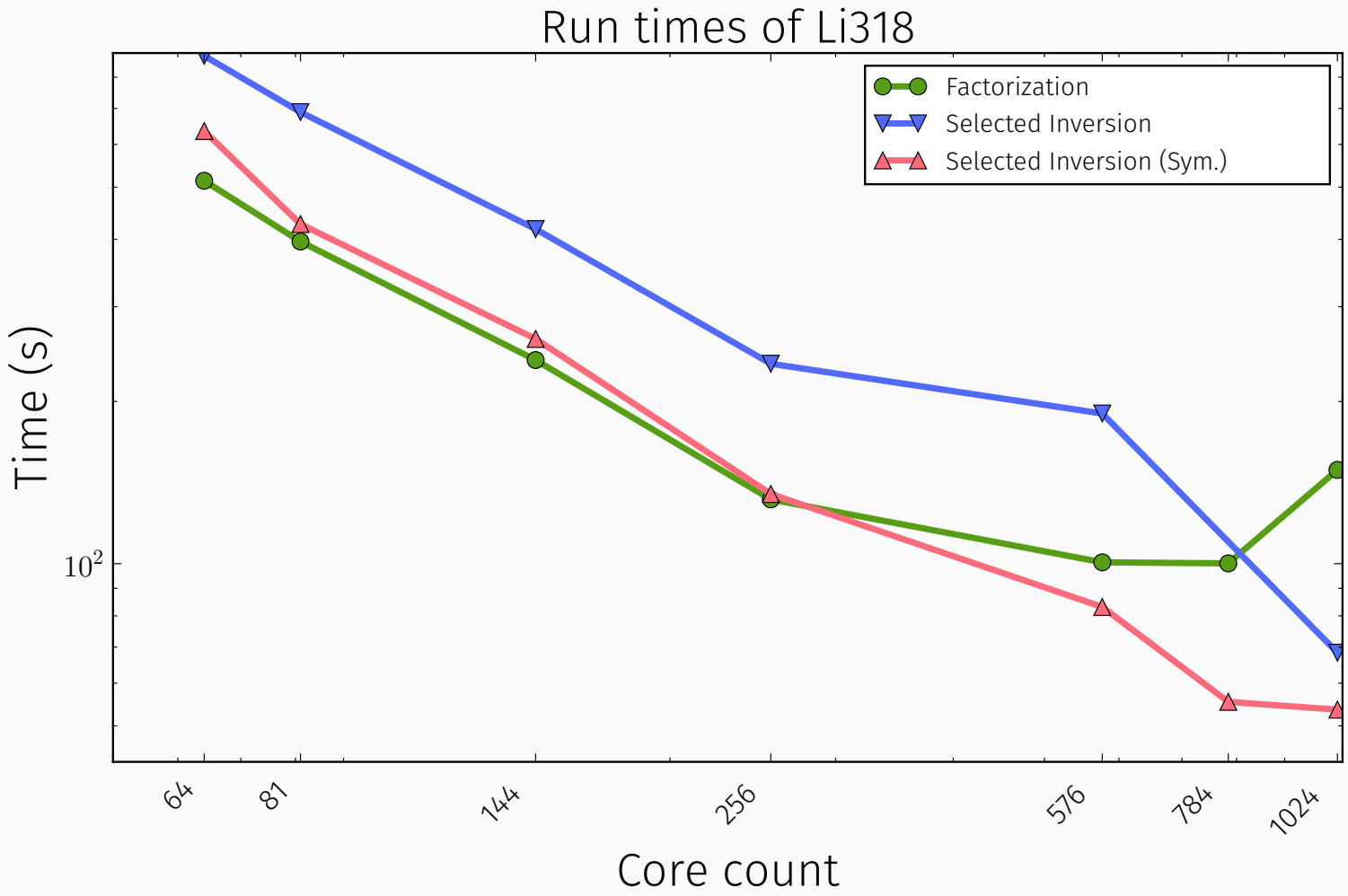
Focus on pipelining computations

STRONG SCALING: LI318_222



n=102,400 , nnz=402,560,000 , nnz(L+U)=5.1×10^9

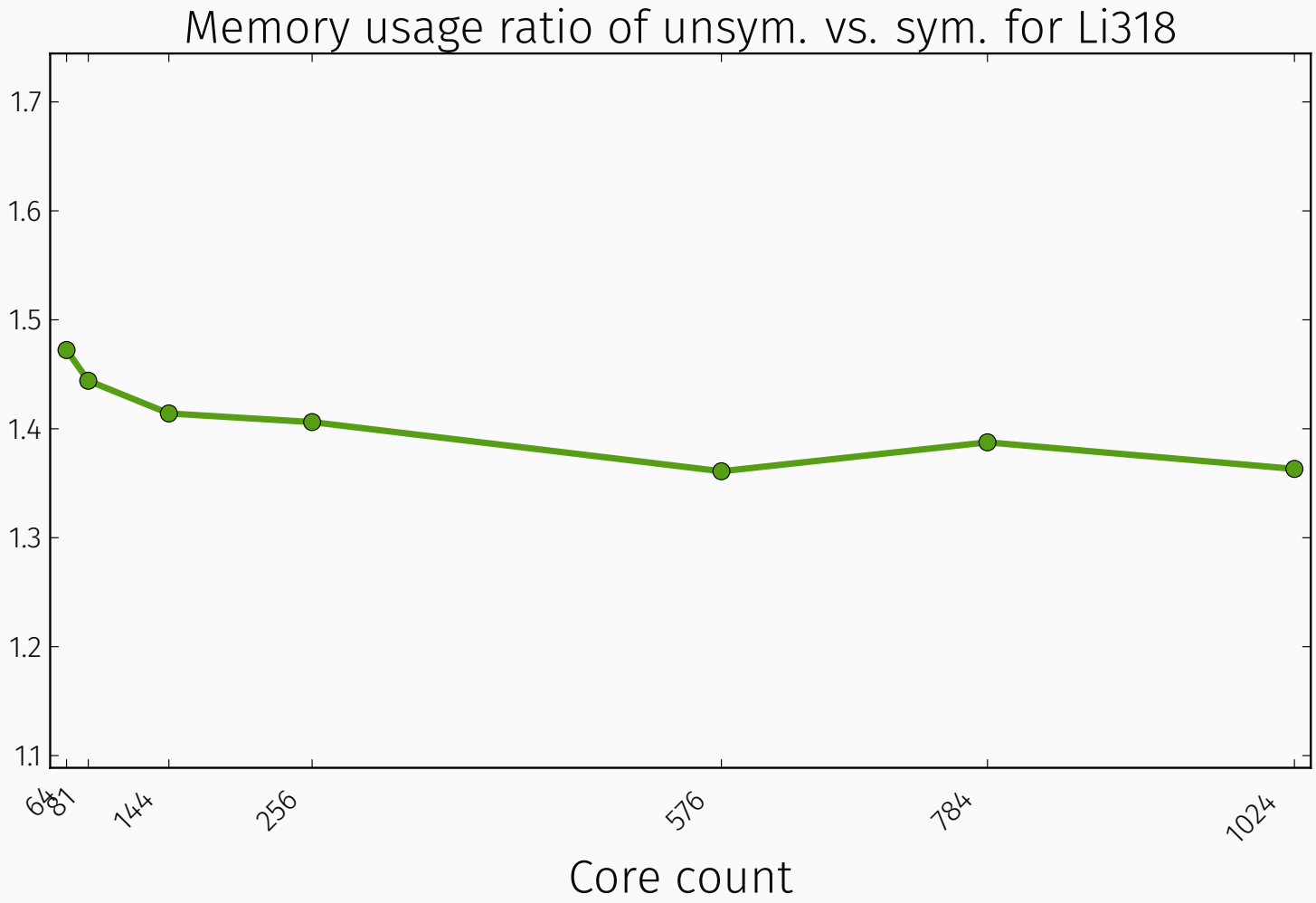
STRONG SCALING: LI318_222



n=102,400 , nnz=402,560,000 , nnz(L+U)=5.1×10^9

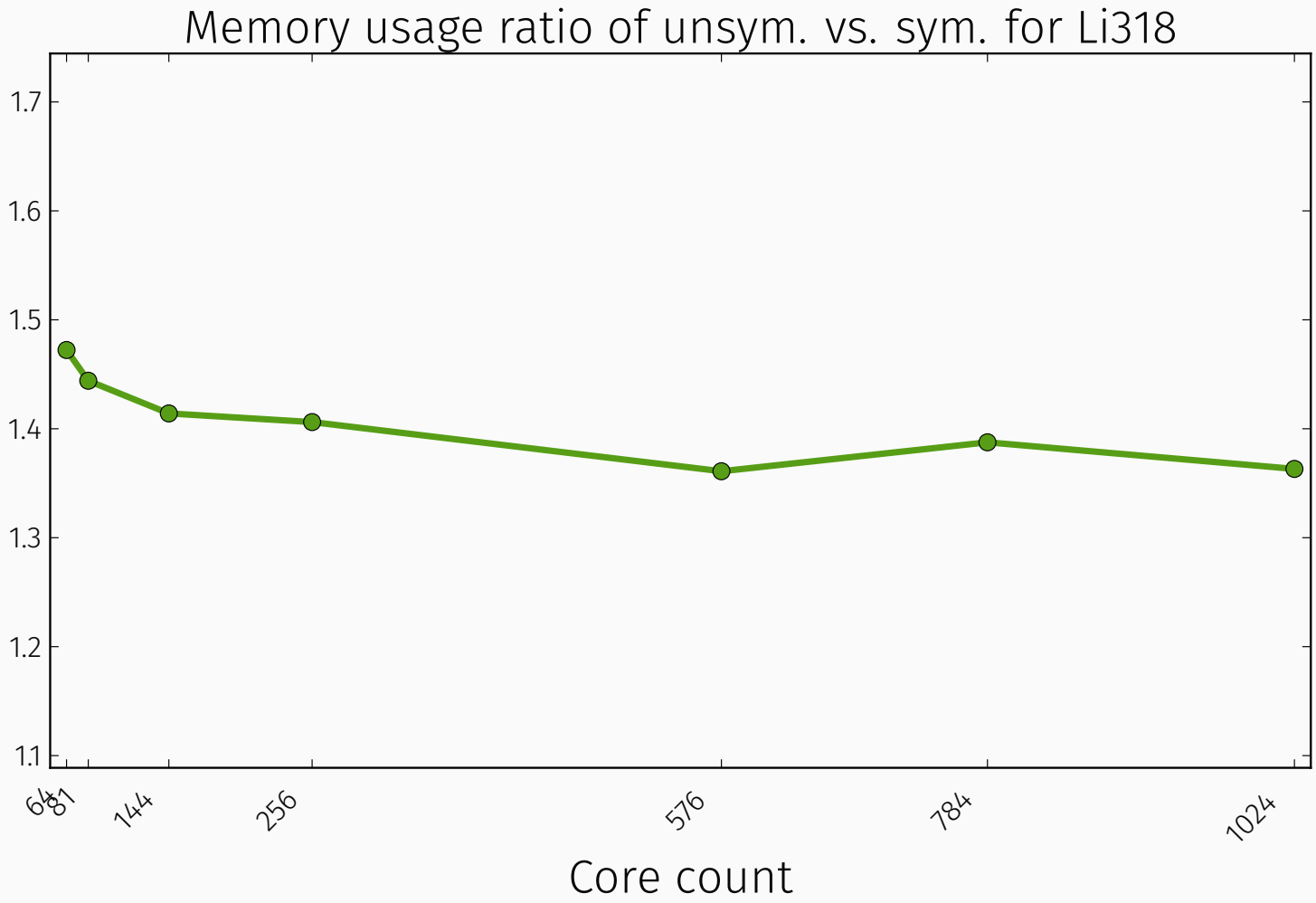
Finer granularity and asynchrony beneficial

MEMORY USAGE: LI318_222



n=102,400 , nnz=402,560,000 , nnz(L+U)=5.1×10^9

MEMORY USAGE: LI318_222



n=102,400 , nnz=402,560,000 , nnz(L+U)=5.1×10^9

For a non symmetric factorization, memory usage lowered by 30%

OUTLINE

Left-looking and Right-looking selected inversion

- Left-looking and Right-looking factorizations
- Right-looking has good properties for parallel systems

Left-looking and Right-looking selected inversion

- Left-looking and Right-looking factorizations
- Right-looking has good properties for parallel systems
- Left-looking and Right-looking Selected Inversion

Left-looking and Right-looking Selected Inversion

- Left-looking and Right-looking factorizations
- Right-looking has good properties for parallel systems
- Right-looking and **Left-looking** Selected Inversion

Left-looking and Right-looking Selected Inversion

- Left-looking and Right-looking factorizations
- Right-looking has good properties for parallel systems
- Right-looking and **Left-looking** Selected Inversion
- Order in which updates are computed

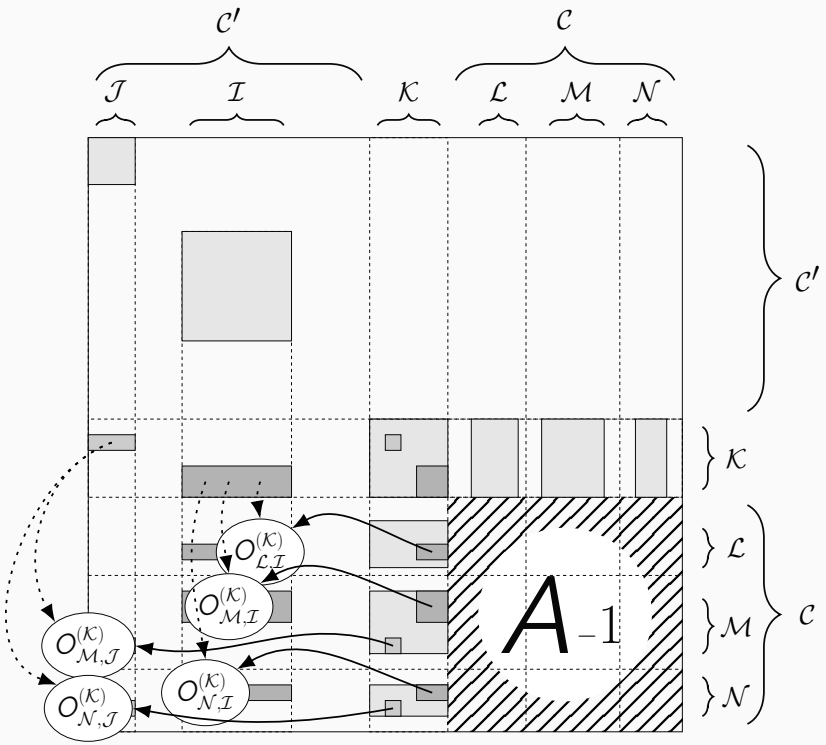
Left-looking and Right-looking Selected Inversion

- Left-looking and Right-looking factorizations
- Right-looking has good properties for parallel systems
- Right-looking and **Left-looking** Selected Inversion
- Order in which updates are computed
- Task approach

Left-looking and Right-looking Selected Inversion

- Left-looking and Right-looking factorizations
- Right-looking has good properties for parallel systems
- Right-looking and **Left-looking** Selected Inversion
- Order in which updates are computed
- Task approach
- Left-looking: Outer-product and Inner-product phases

OUTER-PRODUCT PHASE

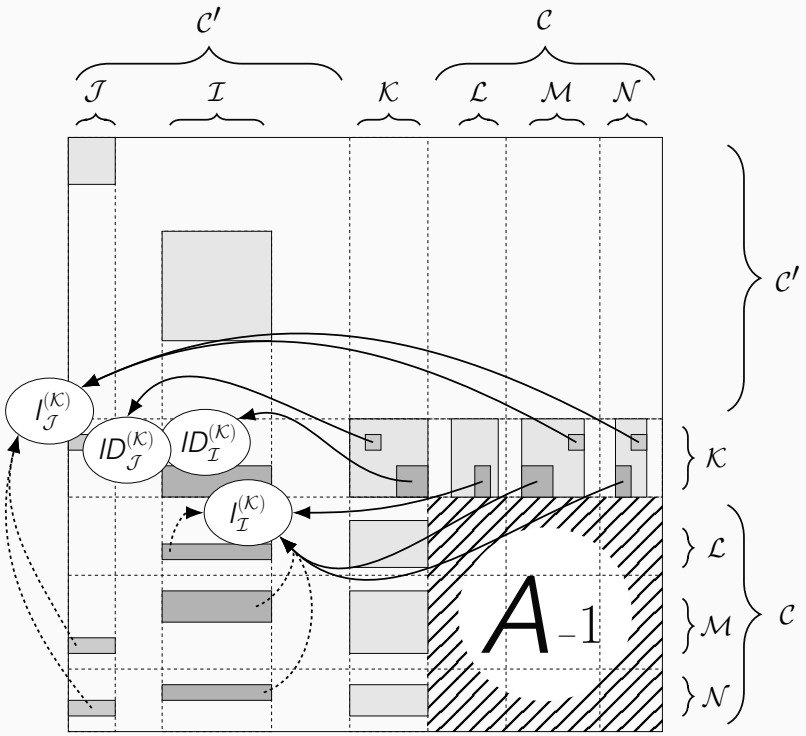


Outer-product task parallelism:

- $O^{(\mathcal{K})}_{*,\mathcal{I}}$ and $O^{(\mathcal{K})}_{*,\mathcal{J}}$ are outer-product updates from $\mathcal{K}$ to $\mathcal{I}$ and $\mathcal{J}$ in $\mathcal{C}'$.

- Data dependencies from values of A^{-1} are denoted with solid arrows.
- Data dependencies from values in LU factors are indicated using dashed arrows.

INNER-PRODUCT PHASE

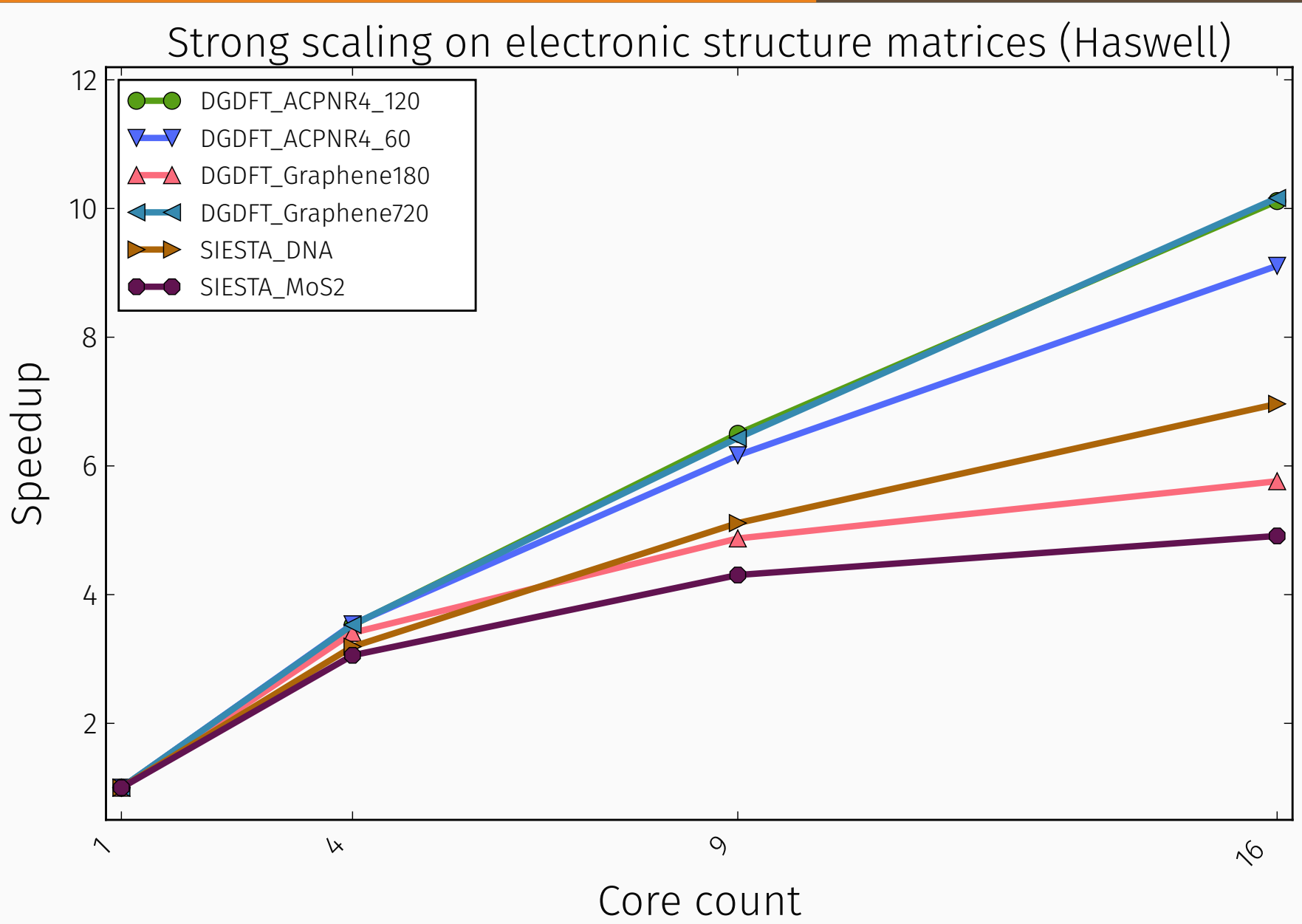


Inner-product and update from diagonal task parallelism:

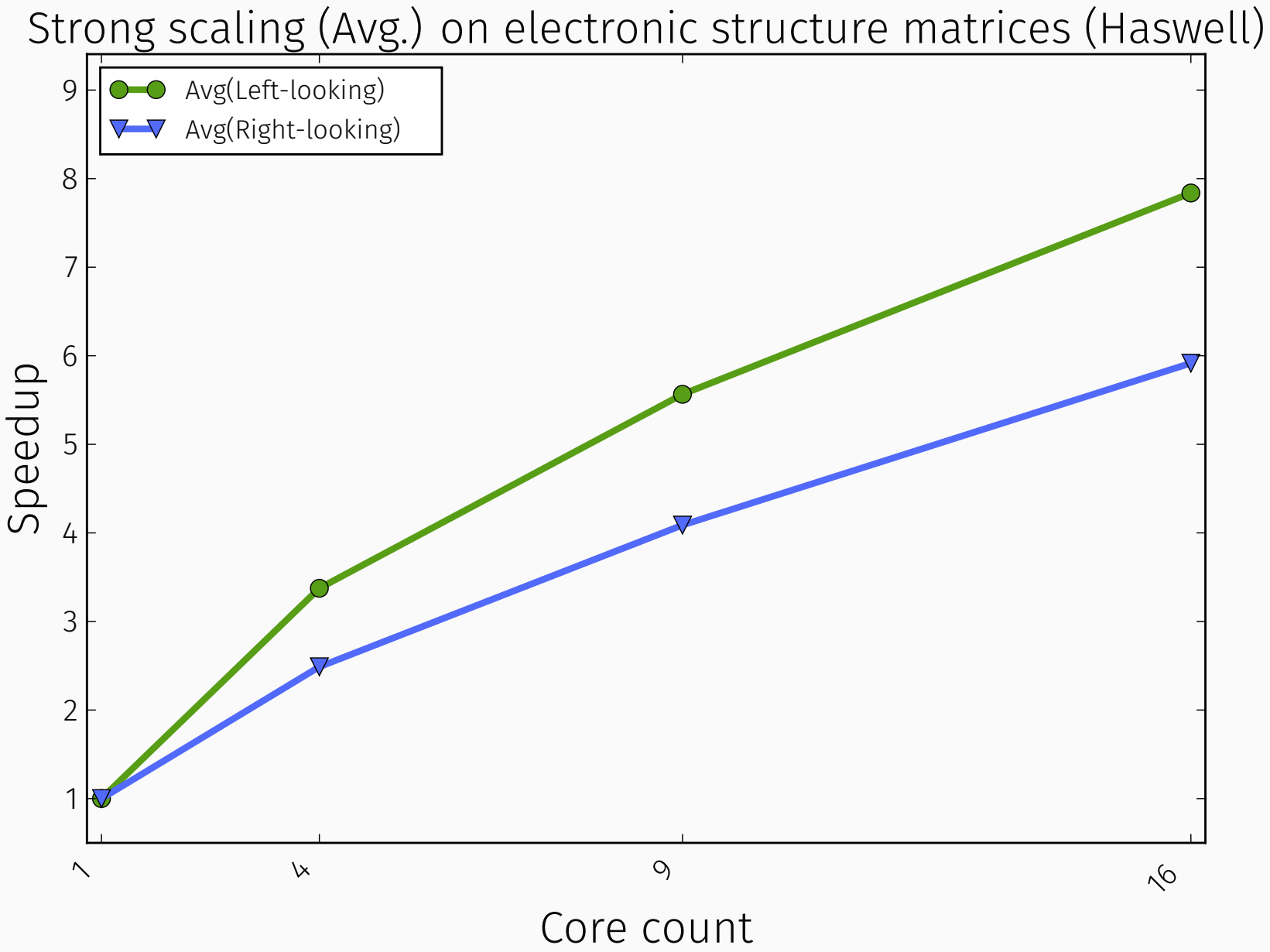
- $I_{\mathcal{I}}^{(\mathcal{K})}$ and $I_{\mathcal{J}}^{(\mathcal{K})}$ are inner-product updates from $\mathcal{K}$ to $\mathcal{I}$ and $\mathcal{J}$ in $\mathcal{C}'$.
- $ID_{\mathcal{I}}^{(\mathcal{K})}$ and $ID_{\mathcal{J}}^{(\mathcal{K})}$ are updates from diagonal block of $\mathcal{K}$.

- Data dependencies from values of A^{-1} are denoted with solid arrows.
- Data dependencies from values in LU factors are indicated using dashed arrows.

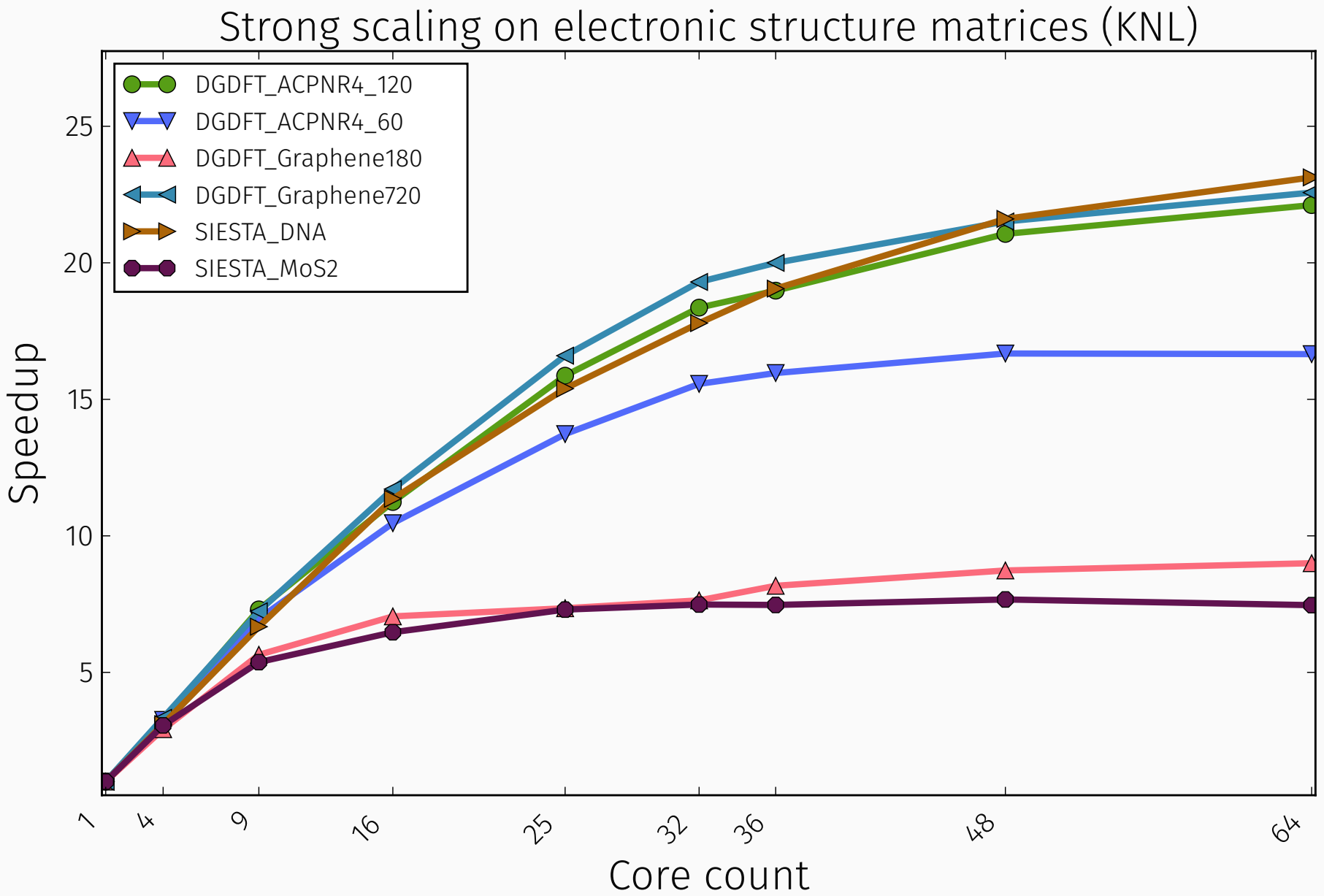
SPEEDUPS ACHIEVED BY LEFT-LOOKING ON CORI HASWELL



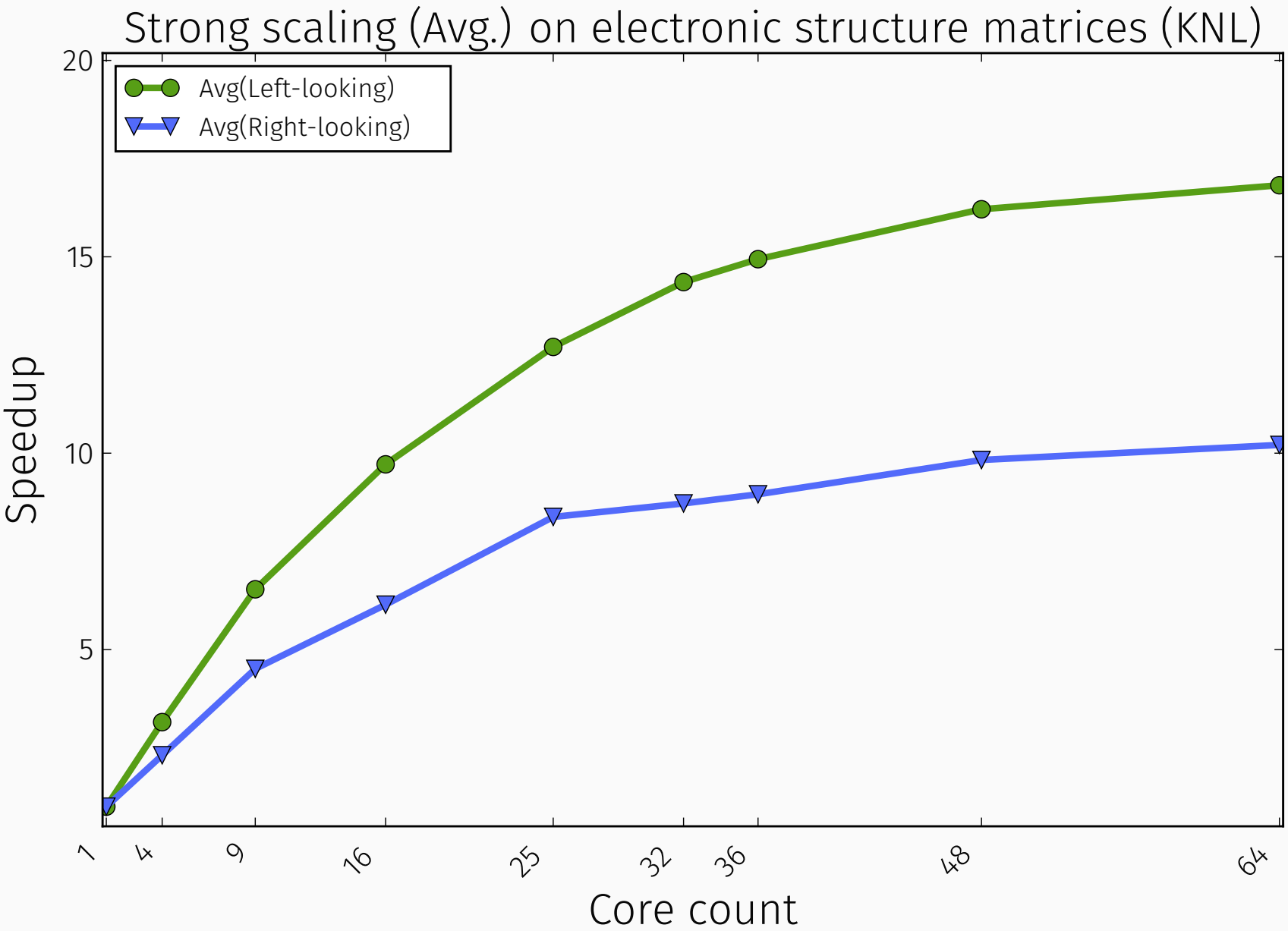
AVG. SPEEDUPS ACHIEVED ON CORI HASWELL



SPEEDUPS ACHIEVED BY LEFT-LOOKING ON CORI KNL



AVG. SPEEDUPS ACHIEVED ON CORI KNL



OUTLINE

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = √A_{j,j}
    for row i = j + 1 to n do
        ℓ_{i,j} = A_{i,j} / ℓ_{j,j}
    end

    for column k = j + 1 to n do
        for row i = k to n do
            A_{i,k} = A_{i,k} − ℓ_{i,j} · ℓ_{k,j}
        end

    end
end
```

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = √A_{j,j}                          } Factor column j
    for row i = j + 1 to n do                   }
        ℓ_{i,j} = A_{i,j} / ℓ_{j,j}             }
    end                                         }

    for column k = j + 1 to n do
        for row i = k to n do
            A_{i,k} = A_{i,k} − ℓ_{i,j} · ℓ_{k,j}
        end

    end
end
```

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = √A_{j,j}                                   } Factor column j
    for row i = j + 1 to n do                            }
        ℓ_{i,j} = A_{i,j} / ℓ_{j,j}                      }
    end                                                  }

    for column k = j + 1 to n do                         } Update next columns
        for row i = k to n do                            }
            A_{i,k} = A_{i,k} − ℓ_{i,j} · ℓ_{k,j}        }
        end                                              }
    end
end
```

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = √A_{j,j}                                  } Factor column j
    for row i = j + 1 to n do                           }
        ℓ_{i,j} = A_{i,j} / ℓ_{j,j}                     }
    end                                                 }
    for column k = j + 1 to n do                        } Update next columns
        for row i = k to n do                           } and Aggregate updates
            A_{i,k} = A_{i,k} − ℓ_{i,j} · ℓ_{k,j}       }
        end                                             }
    end
end
```

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = √A_{j,j}                                   } Factor column j
    for row i = j + 1 to n do                            }
        ℓ_{i,j} = A_{i,j} / ℓ_{j,j}                      }
    end                                                  }

    for column k = j + 1 to n do                         } Update next columns
        for row i = k to n do                            } and Aggregate updates
            A_{i,k} = A_{i,k} − ℓ_{i,j} · ℓ_{k,j}        } for row i = k to n do
        end                                              }     tmp_i = tmp_i + ℓ_{i,j} · ℓ_{k,j}
    end                                                  } end
                                                         } A_{*,k} = A_{*,k} − tmp_*
end
```

fan-both ALGORITHM

- Three families [Ashcraft'95]:
  - Fan-In: "fanning-in updates"
    - Reduce **aggregate vectors** (updates)
    - Factorize column
    - Compute all updates from *that column* locally

fan-both ALGORITHM

- Three families [Ashcraft'95]:
  - Fan-In: "fanning-in updates"
    - Reduce **aggregate vectors** (updates)
    - Factorize column
    - Compute all updates from *that column* locally
  - Fan-Out: "fanning-out factors"
    - Factorize column
    - Distribute the Cholesky factor
    - Compute and apply all updates to *my column*.

fan-both ALGORITHM

- Three families [Ashcraft'95]:
  - Fan-In: "fanning-in updates"
    - Reduce **aggregate vectors** (updates)
    - Factorize column
    - Compute all updates from *that column* locally
  - Fan-Out: "fanning-out factors"
    - Factorize column
    - Distribute the Cholesky factor
    - Compute and apply all updates to *my column*.

Family determined by type of data exchanged

fan-both ALGORITHM

- Three families [Ashcraft'95]:
  - Fan-In: "fanning-in updates"
    - Reduce **aggregate vectors** (updates)
    - Factorize column
    - Compute all updates from *that column* locally
  - Fan-Out: "fanning-out factors"
    - Factorize column
    - Distribute the Cholesky factor
    - Compute and apply all updates to *my column*.

Family determined by type of data exchanged

Fan-In, Fan-Out $\subset$ Fan-Both

DESIGN PRINCIPLES

- Remove synchronization points
  - Asynchronous point to point send

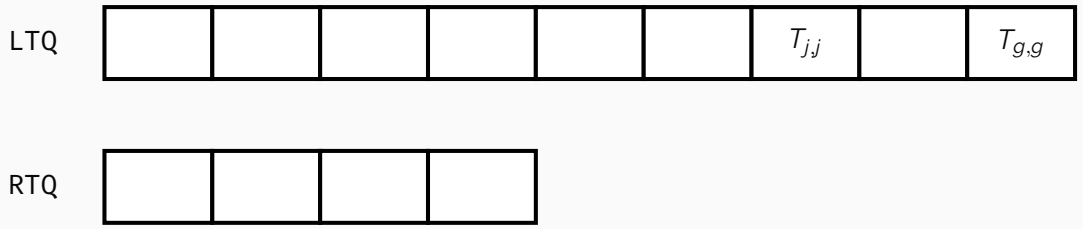
- Remove synchronization points
  - Asynchronous point to point send
  - Non-collectives = full asynchronicity

DESIGN PRINCIPLES

- Remove synchronization points
  - Asynchronous point to point send
  - Non-collectives = full asynchronicity
- Minimize memory operations
  - Row-major layout
  - Avoid making extra copies when sending data

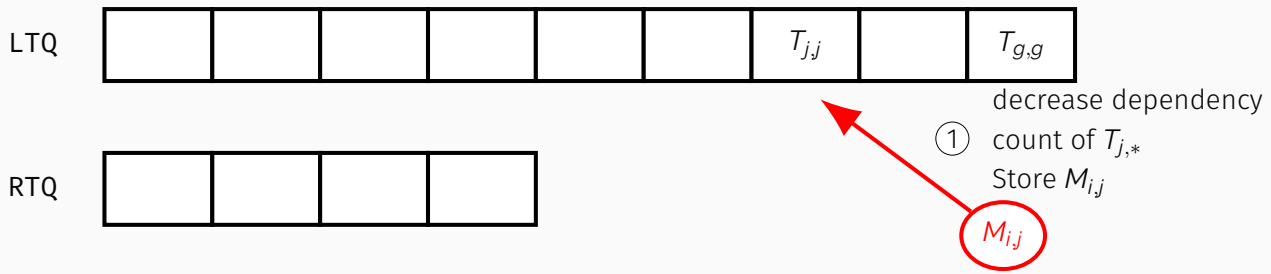
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**



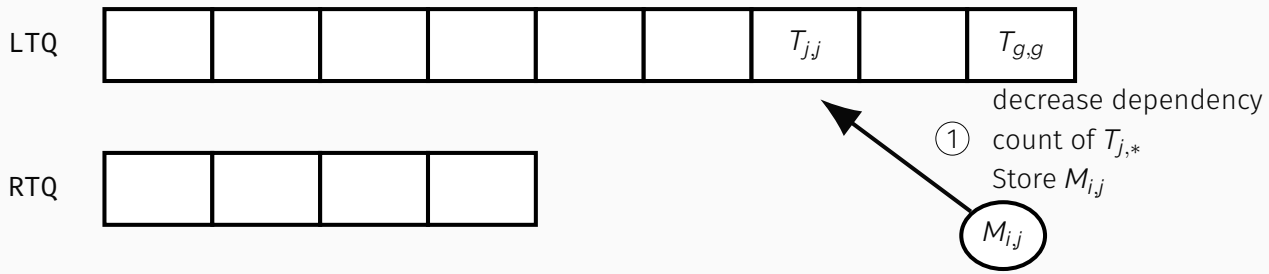
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**
  - Dependency count



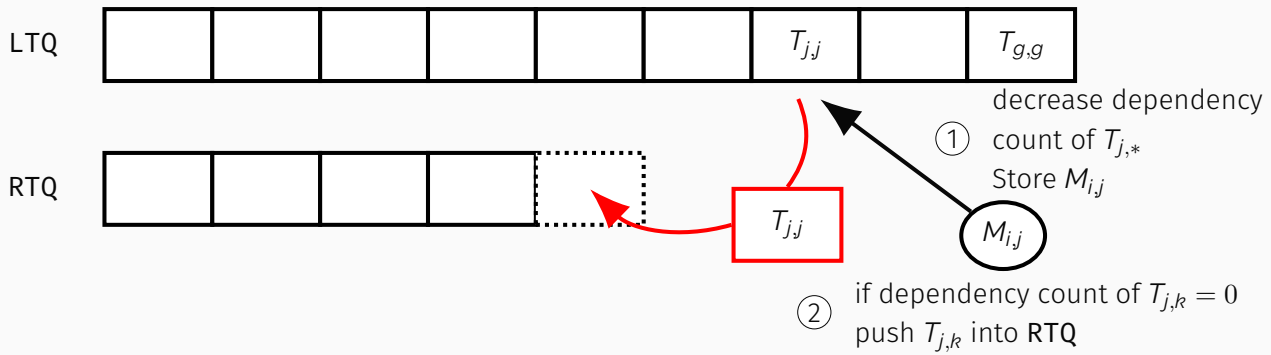
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**
  - Dependency count



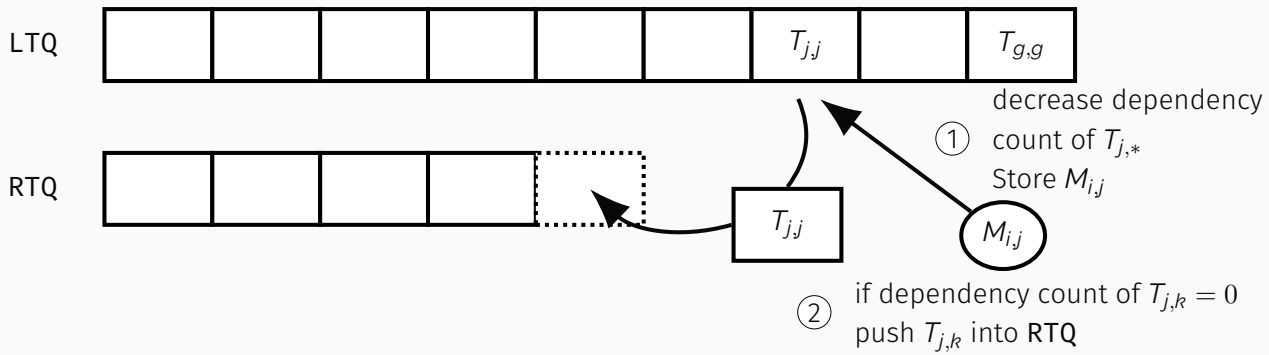
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**
  - Dependency count
  - Ready tasks are placed in **RTQ**



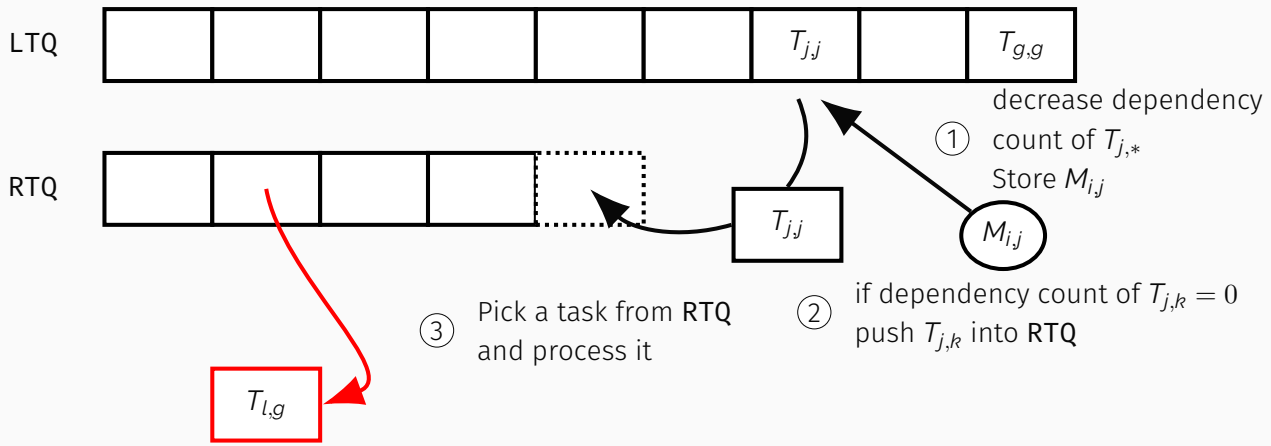
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**
  - Dependency count
  - Ready tasks are placed in **RTQ**



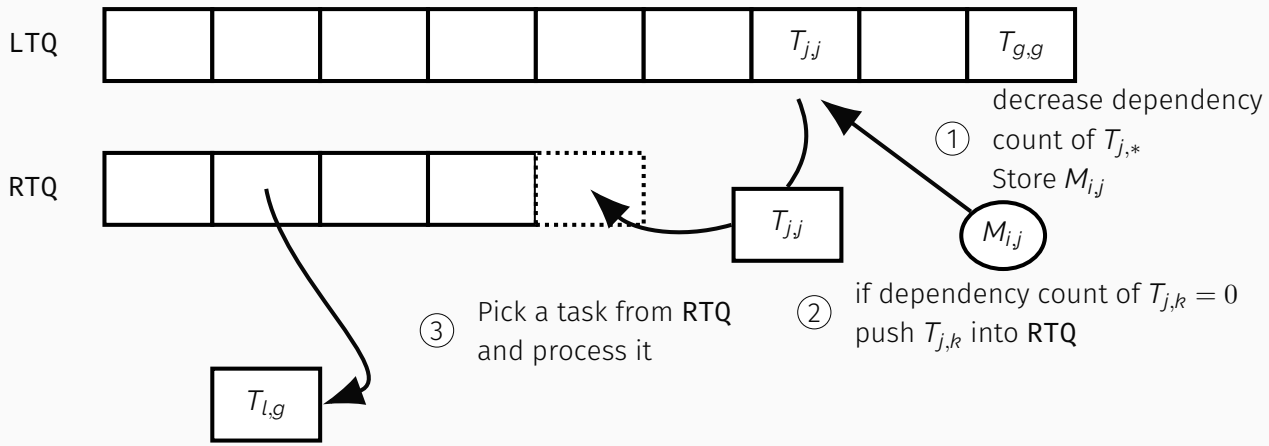
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**
  - Dependency count
  - Ready tasks are placed in **RTQ**



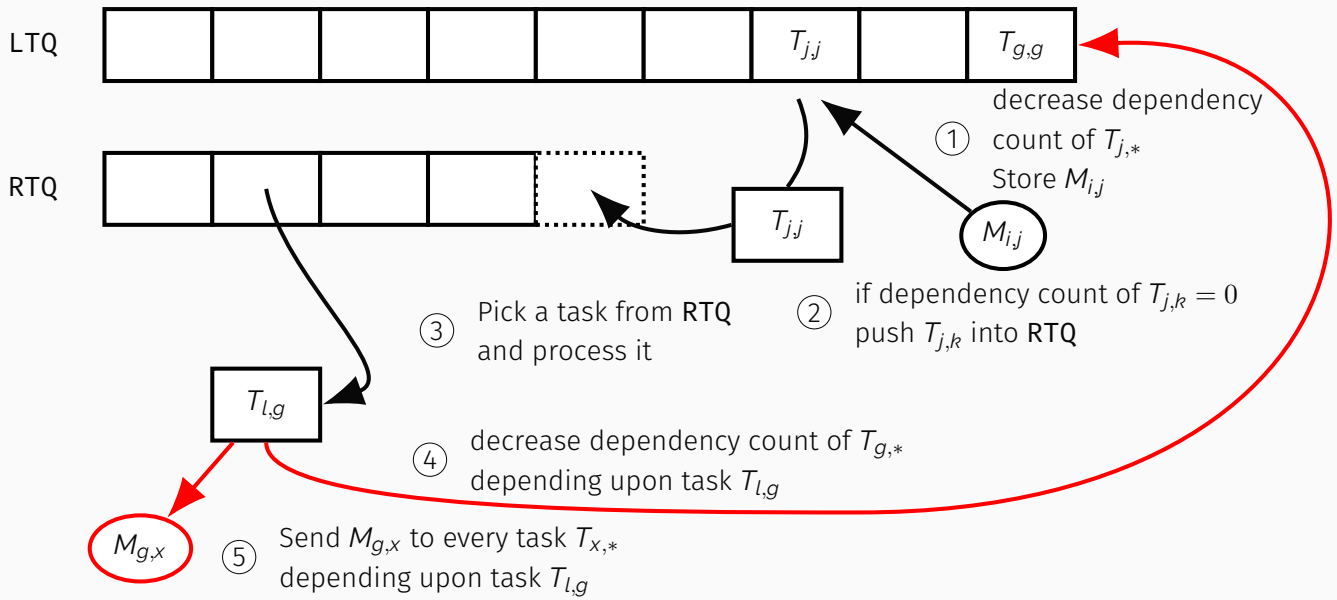
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**
  - Dependency count
  - Ready tasks are placed in **RTQ**



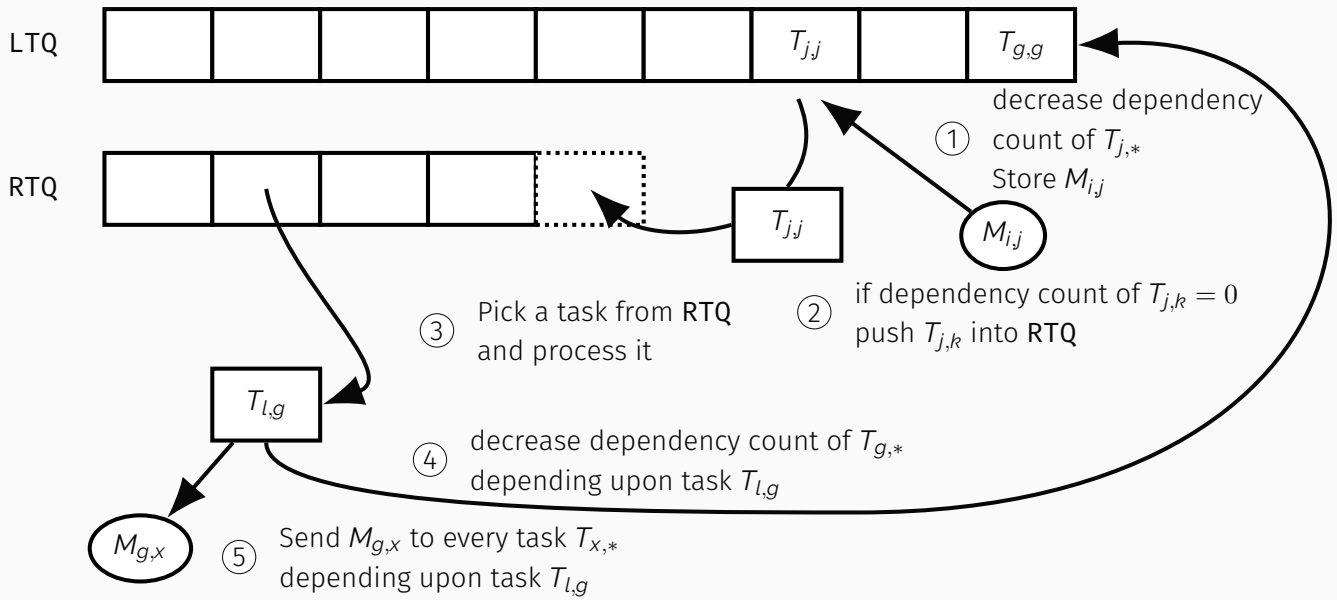
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**
  - Dependency count
  - Ready tasks are placed in **RTQ**



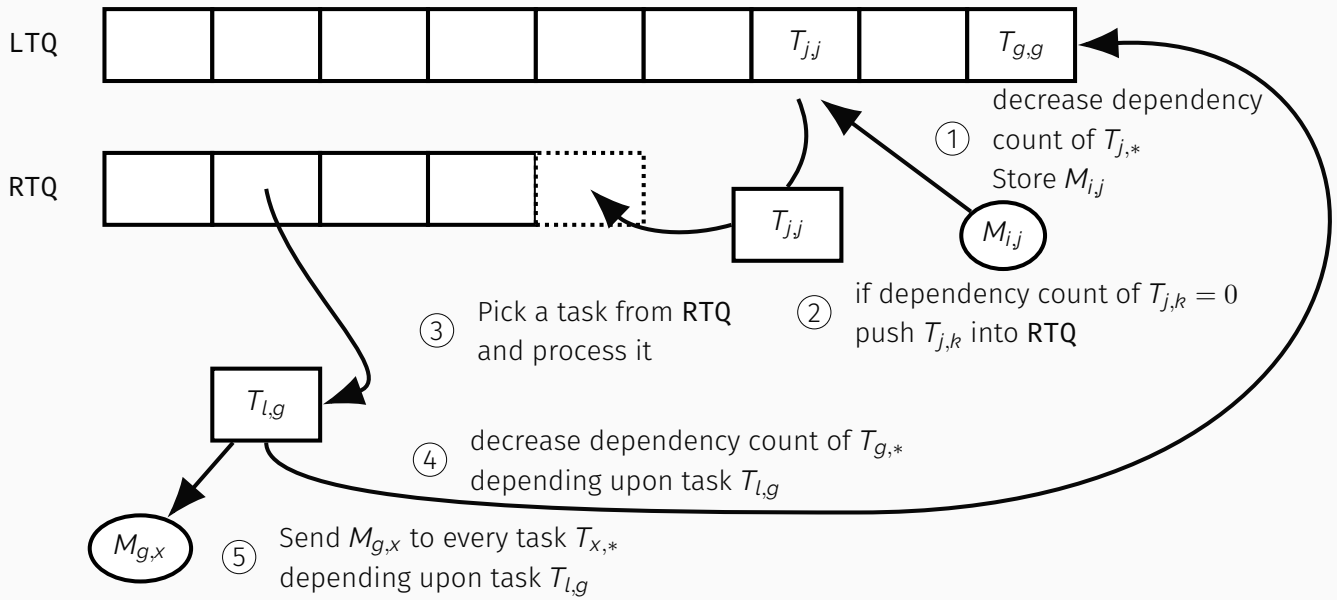
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**
  - Dependency count
  - Ready tasks are placed in **RTQ**



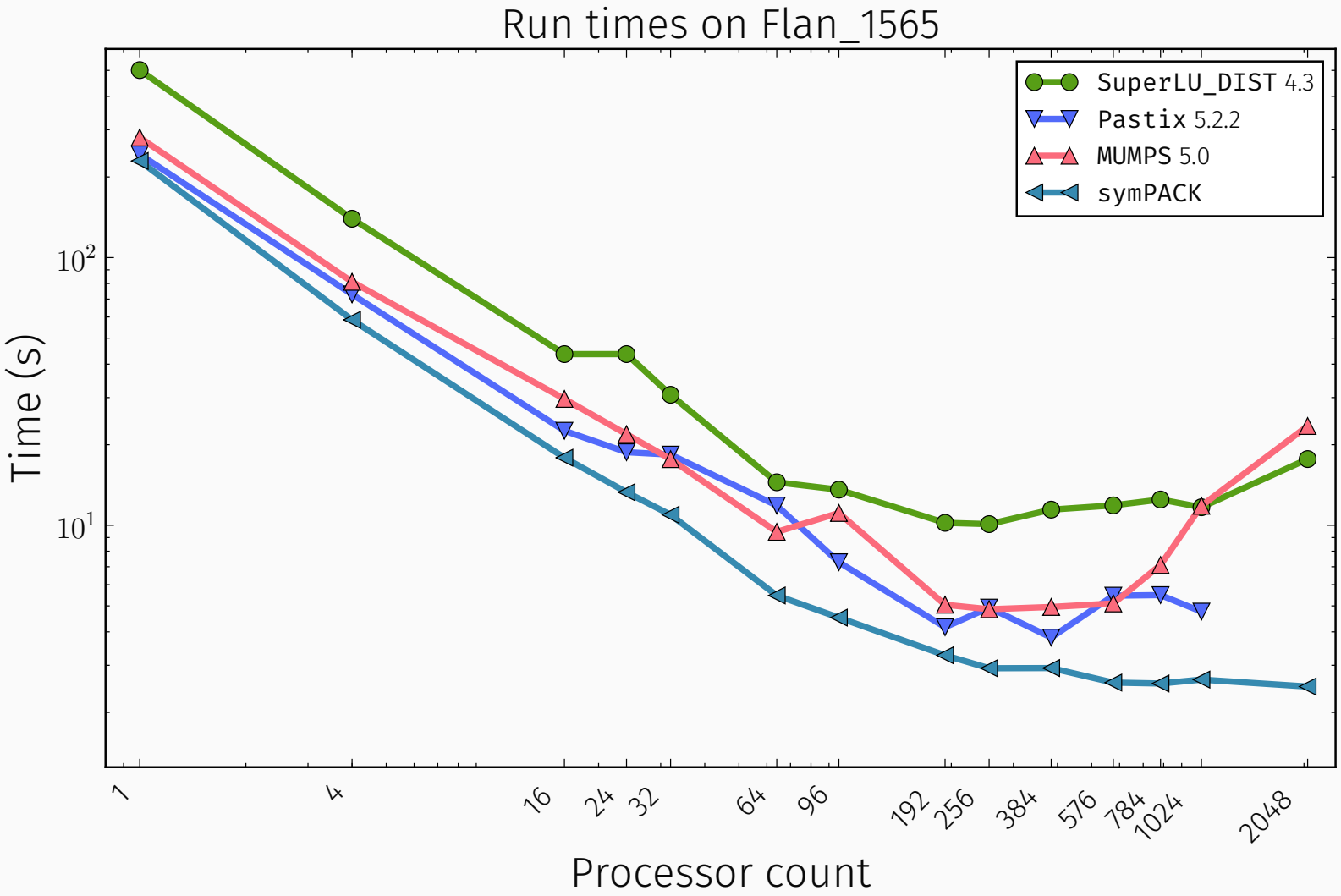
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue **LTQ**
  - Dependency count
  - Ready tasks are placed in **RTQ**



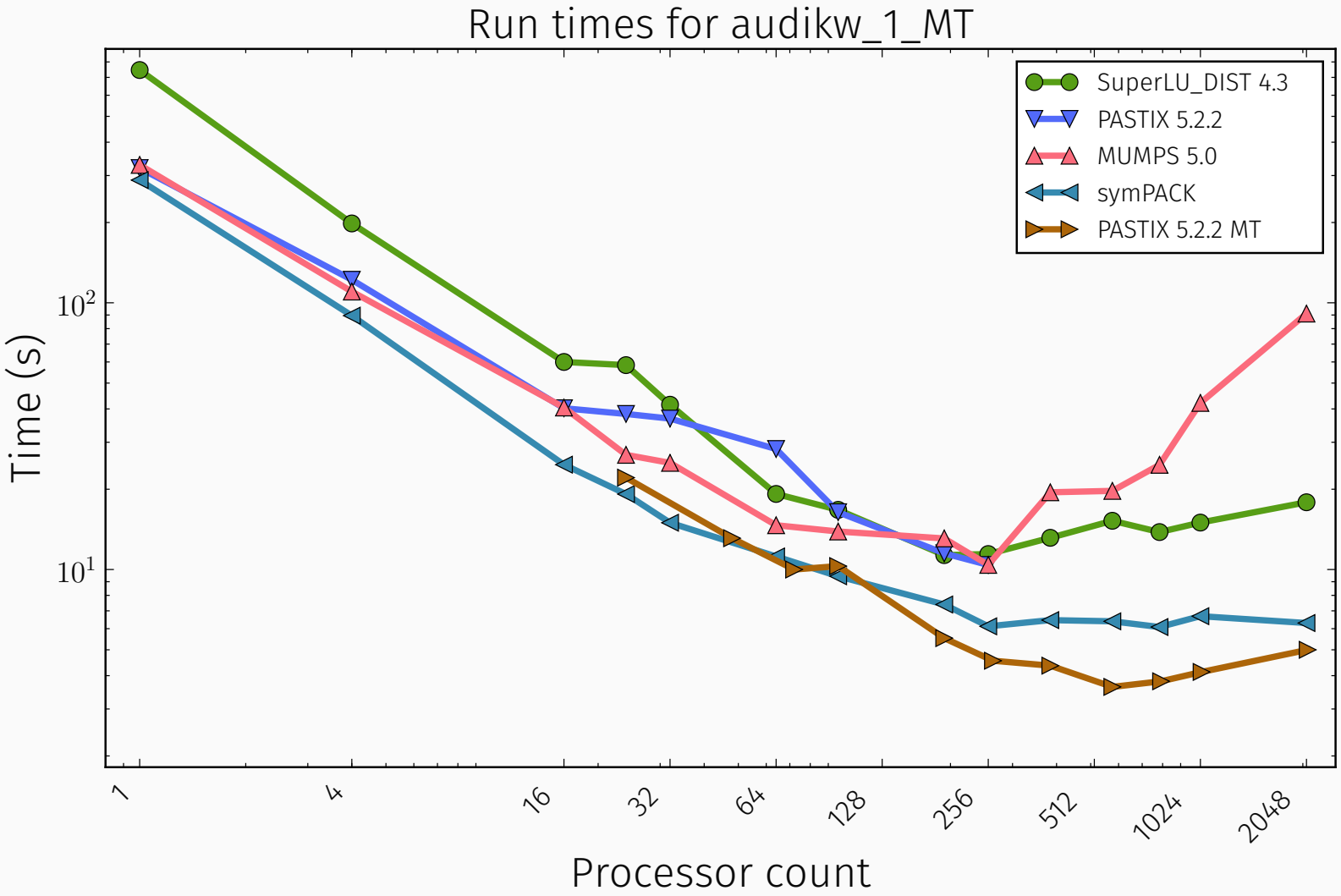
Scheduling policy ? FIFO, close to diagonal, etc.

STRONG SCALING VS. STATE-OF-THE-ART (NERSC EDISON)



n=1,564,794 nnz(A)=57,865,083 nnz(L)=1,574,541,576

STRONG SCALING VS. STATE-OF-THE-ART



n=943,695 nnz(A)=39,297,771 nnz(L)=1,221,674,796

OUTLINE

CONCLUSION

- Parallel Selected Inversion with symmetric storage
- Pipelining and asynchronous task execution model are critical for performance

CONCLUSION

- Parallel Selected Inversion with symmetric storage
- Pipelining and asynchronous task execution model are critical for performance
- `PSelInv` available in the PEXSI library

`http://www.pexsi.org/`

CONCLUSION

- Parallel Selected Inversion with symmetric storage
- Pipelining and asynchronous task execution model are critical for performance
- `PSelInv` available in the PEXSI library

  `http://www.pexsi.org/`

- Highly scalable left-looking shared-memory implementation

CONCLUSION

- Parallel Selected Inversion with symmetric storage
- Pipelining and asynchronous task execution model are critical for performance
- `PSelInv` available in the PEXSI library

  `http://www.pexsi.org/`

- Highly scalable left-looking shared-memory implementation (available in future release)

CONCLUSION

- Parallel Selected Inversion with symmetric storage
- Pipelining and asynchronous task execution model are critical for performance
- `PSelInv` available in the PEXSI library

  `http://www.pexsi.org/`

- Highly scalable left-looking shared-memory implementation (available in future release)
- New symmetric solver `symPACK`
- One sided approach using UPC++
- Asynchronous task execution model & dynamic scheduling
- `symPACK` available at:

  `http://www.sympack.org/`