SPARSE CHOLESKY FACTORIZATION USING A FAN-BOTH APPROACH

Esmond G. Ng

Computational Research Division
Lawrence Berkeley National Labatory

Joint work with Mathias Jacquelin*, Kathy Yelick and Yili Zheng

May 13, 2016

- The **Applied Mathematics Department** has 4 groups.

- The **Applied Mathematics Department** has 4 groups.
    - Applied Numerical Algorithms (Phil Colella) – Develop advanced numerical algorithms & software for PDEs, with the application of the software to scientific and engineering problems.

- The **Applied Mathematics Department** has 4 groups.
  - Applied Numerical Algorithms (Phil Colella) – Develop advanced numerical algorithms & software for PDEs, with the application of the software to scientific and engineering problems.
  - Center for Computational Sciences & Engineering (Ann Almgren) – Develop and apply advanced computational methodologies to solve large-scale scientific and engineering problems.

- The **Applied Mathematics Department** has 4 groups.
  - Applied Numerical Algorithms (Phil Colella) – Develop advanced numerical algorithms & software for PDEs, with the application of the software to scientific and engineering problems.
  - Center for Computational Sciences & Engineering (Ann Almgren) – Develop and apply advanced computational methodologies to solve large-scale scientific and engineering problems.
  - Mathematics Group (Jamie Sethian) – Develop new mathematical models, devise new algorithms, and explore new applications.

- The **Applied Mathematics Department** has 4 groups.
  - Applied Numerical Algorithms (Phil Colella) – Develop advanced numerical algorithms & software for PDEs, with the application of the software to scientific and engineering problems.
  - Center for Computational Sciences & Engineering (Ann Almgren) – Develop and apply advanced computational methodologies to solve large-scale scientific and engineering problems.
  - Mathematics Group (Jamie Sethian) – Develop new mathematical models, devise new algorithms, and explore new applications.
  - Scalable Solvers (Sherry Li) – Develop efficient algebraic solvers and fast, scalable, library implementations.

- The **Applied Mathematics Department** has 4 groups.
  - Applied Numerical Algorithms (Phil Colella) – Develop advanced numerical algorithms & software for PDEs, with the application of the software to scientific and engineering problems.
  - Center for Computational Sciences & Engineering (Ann Almgren) – Develop and apply advanced computational methodologies to solve large-scale scientific and engineering problems.
  - Mathematics Group (Jamie Sethian) – Develop new mathematical models, devise new algorithms, and explore new applications.
  - Scalable Solvers (Sherry Li) – Develop efficient algebraic solvers and fast, scalable, library implementations.
- CAMERA: Center for Advanced Mathematics for Energy Research Applications (Jamie Sethian).

- Given an $n \times n$ matrix A, compute a triangular factorization: $A \rightarrow LU$, where L is lower triangular and U is upper triangular.

MOTIVATION AND OBJECTIVE

- Given an $n \times n$ matrix A, compute a triangular factorization: $A \rightarrow LU$, where L is lower triangular and U is upper triangular.

- Triangular factorization of a matrix is useful ...
  - Solving linear systems with many right-hand sides.
  - Solving ill-conditioned linear systems.
    - e.g., using shift-invert Lanczos to compute eigenvalues.
  - Computing elements of the inverse of a matrix.

- Given an $n \times n$ matrix A, compute a triangular factorization: $A \rightarrow LU$, where L is lower triangular and U is upper triangular.
- Triangular factorization of a matrix is useful ...
  - Solving linear systems with many right-hand sides.
  - Solving ill-conditioned linear systems.
    - e.g., using shift-invert Lanczos to compute eigenvalues.
  - Computing elements of the inverse of a matrix.
- Focus on **large sparse symmetric positive definite matrices**.

CHALLENGES IN PARALLEL SPARSE MATRIX FACTORIZATIONS

- Factorization of a sparse matrix produces fill.
  - ⇒ Higher memory requirement.
  - ⇒ Higher core count & higher communication cost.
- Irregular sparsity structure.
  - ⇒ irregular communication pattern.
- Parallelizing sparse matrix factorization can be hard.

- Open-source parallel sparse symmetric factorization codes exist ...
  - `MUMPS` (Université de Toulouse).
  - `PaStiX` (INRIA Bordeaux).
- They may have scalability issues even at modest core counts.

STATE-OF-THE-ART PARALLEL SPARSE SYMMETRIC FACTORIZATION CODES

- Open-source parallel sparse symmetric factorization codes exist ...
  - `MUMPS` (Université de Toulouse).
  - `PaStiX` (INRIA Bordeaux).
- They may have scalability issues even at modest core counts.
- Our goal is to develop a more scalable sparse symmetric factorization code with reduced communication.

- Basic algorithm for computing $A = LL^T$ (ignoring sparsity):

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = sqrt(A_{j,j})
    for row k = j + 1 to n do
        ℓ_{k,j} = A_{k,j} / ℓ_{j,j}
    end

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

    end
end
```

- Basic algorithm for computing $A = LL^T$ (ignoring sparsity):

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = sqrt(A_{j,j})                  ⎫
    for row k = j + 1 to n do                ⎪
        ℓ_{k,j} = A_{k,j} / ℓ_{j,j}          ⎬ Factor column j
    end                                      ⎭

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

- Basic algorithm for computing $A = LL^T$ (ignoring sparsity):

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = sqrt(A_{j,j})                      ⎫
    for row k = j + 1 to n do                    ⎬ Factor column j
        ℓ_{k,j} = A_{k,j} / ℓ_{j,j}              ⎪
    end                                          ⎭

    for column i = j + 1 to n do                 ⎫ Update remaining columns
        for row k = i to n do                    ⎪
            A_{k,i} = A_{k,i} − ℓ_{i,j} · ℓ_{k,j} ⎬
        end                                      ⎪
    end                                          ⎭
end
```

$\ell_{j,j} = \sqrt{A_{j,j}}$; $\ell_{k,j} = A_{k,j}/\ell_{j,j}$; $A_{k,i} = A_{k,i} - \ell_{i,j} \cdot \ell_{k,j}$

CHOLESKY FACTORIZATION

- Basic algorithm for computing $A = LL^T$ (ignoring sparsity):

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = √A_{j,j}                          ⎫
    for row k = j + 1 to n do                    ⎬ Factor column j
        ℓ_{k,j} = A_{k,j} / ℓ_{j,j}             ⎪
    end                                          ⎭

    for column i = j + 1 to n do                 ⎫ Update remaining columns
        for row k = i to n do                    ⎪
            A_{k,i} = A_{k,i} − ℓ_{i,j} · ℓ_{k,j} ⎪ In a parallel setting, where
        end                                      ⎬ and when updates occur de-
                                                 ⎪ pend on the algorithm for-
    end                                          ⎭ mulation
end
```

- Two classes of parallel algorithms: `Fan-In` and `Fan-Out`.

- Two classes of parallel algorithms: `Fan-In` and `Fan-Out`.
- `Fan-In` factorization:
  - Receive and apply updates.
  - Factorize column.
  - Compute and distribute all updates from that column.

- Two classes of parallel algorithms: `Fan-In` and `Fan-Out`.

- `Fan-In` factorization:
  - Receive and apply updates.
  - Factorize column.
  - Compute and distribute all updates from that column.

- `Fan-Out` factorization:
  - Receive prior factor columns; Compute and apply updates.
  - Factorize column.
  - Distribute factor column.
  - Perform updates on locally-owned columns.

PARALLEL MATRIX FACTORIZATIONS

- Two classes of parallel algorithms: `Fan-In` and `Fan-Out`.
- `Fan-In` factorization:
  - Receive and apply updates.
  - Factorize column.
  - Compute and distribute all updates from that column.
- `Fan-Out` factorization:
  - Receive prior factor columns; Compute and apply updates.
  - Factorize column.
  - Distribute factor column.
  - Perform updates on locally-owned columns.
- Both classes of algorithms are mathematically equivalent.
  - The order of operations may be different.
- Updates destined for the same processor should be aggregrated.

- `Fan-In` and `Fan-Out`: Often described in terms of where the **data** (i.e., columns) are located.
  - `Fan-In`: update for a target column is computed on the processor owning the source column.
  - `Fan-Out`: update for a target column is computed on the processor owning the target column.

DATA-CENTRIC & TASK-CENTRIC VIEWS OF THE FACTORIZATION

- `Fan-In` and `Fan-Out`: Often described in terms of where the **data** (i.e., columns) are located.
  - `Fan-In`: update for a target column is computed on the processor owning the source column.
  - `Fan-Out`: update for a target column is computed on the processor owning the target column.

- An alternative: based on **computational tasks**.
  - The update tasks can be executed on any processors; they don't have to be performed on the processor owning the source column (in `Fan-In`) or the target column (in `Fan-Out`).

fan-both algorithms: task-based factorizations

- Proposed by Ashcraft ('95).
  - Focus on where computation is performed instead of where data is placed.

fan-both ALGORITHMS: TASK-BASED FACTORIZATIONS

- Proposed by Ashcraft ('95).
  - Focus on where computation is performed instead of where data is placed.
- Require a **computational map** to indicate where the computations (i.e., tasks) are performed.

`fan-both` ALGORITHMS: TASK-BASED FACTORIZATIONS

- Proposed by Ashcraft ('95).
  - Focus on where computation is performed instead of where data is placed.

- Require a **computational map** to indicate where the computations (i.e., tasks) are performed.
  - $\textbf{map}(\textbf{target}, \textbf{source})$ denotes the processor that updates column **target** using column **source**:

$$A_{*,\text{target}} = A_{*,\text{target}} - \ell_{\text{target},\text{source}} \cdot \ell_{*,\text{source}}$$

  - When $\textbf{target} = \textbf{source}$, $\textbf{map}(\textbf{source}, \textbf{source})$ is the processor that factors column **source**.

fan-in, fan-out, fan-both

- `Fan-Both` is a broad class of task-based parallel matrix factorization algorithms.
  - Depending on choice of **map**, the update operations from/to a given column can potentially be distributed over multiple processors.
    $\Rightarrow$ Increased parallelism.

- Fan-Both is a broad class of task-based parallel matrix factorization algorithms.
  - Depending on choice of **map**, the update operations from/to a given column can potentially be distributed over multiple processors.
    $\Rightarrow$ Increased parallelism.
  - May result in increased communication message count and increased communication volume.
  - Ashcraft showed that, for matrices coming from an $m \times m$ grid and using $\sqrt{P} \times \sqrt{P}$ computational map, each factorization step involves at most $\sqrt{P}$ processors, where P is the total number of processors available.

fan-in, fan-out, fan-both

- **Fan-Both** is a broad class of task-based parallel matrix factorization algorithms.
  - Depending on choice of **map**, the update operations from/to a given column can potentially be distributed over multiple processors.
    $\Rightarrow$ Increased parallelism.
  - May result in increased communication message count and increased communication volume.
  - Ashcraft showed that, for matrices coming from an $m \times m$ grid and using $\sqrt{P} \times \sqrt{P}$ computational map, each factorization step involves at most $\sqrt{P}$ processors, where P is the total number of processors available.
  - **Fan-In** and **Fan-Out** would involve at most P processors at each step.

fan-both, fan-in, fan-out

- Fan-Both includes Fan-In and Fan-Out.
  - Fan-In: $\text{map}(\text{target}, \text{source}) = \text{map}(\text{source}, \text{source})$.
  - Fan-Out: $\text{map}(\text{target}, \text{source}) = \text{map}(\text{target}, \text{target})$.

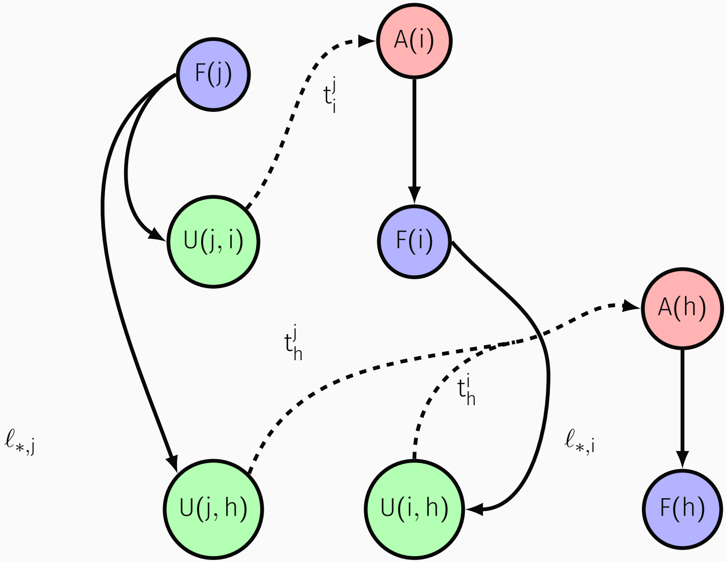
fan-both, fan-in, fan-out

- `Fan-Both` includes `Fan-In` and `Fan-Out`.
  - `Fan-In`: $\text{map}(\text{target}, \text{source}) = \text{map}(\text{source}, \text{source})$.
  - `Fan-Out`: $\text{map}(\text{target}, \text{source}) = \text{map}(\text{target}, \text{target})$.
- `Fan-In` and `Fan-Out` have only one type of messages, but `Fan-Both` has two types of messages.

fan-both, fan-in, fan-out

- Fan-Both includes Fan-In and Fan-Out.
  - Fan-In: $\text{map}(\text{target}, \text{source}) = \text{map}(\text{source}, \text{source})$.
  - Fan-Out: $\text{map}(\text{target}, \text{source}) = \text{map}(\text{target}, \text{target})$.
- Fan-In and Fan-Out have only one type of messages, but Fan-Both has two types of messages.
- The Fan-In and Fan-Out task graphs can be described more compactly by a tree structure, but Fan-Both has a more elaborate task graph.

fan-both TASK GRAPH



- Choice of data distribution.
  - 1D cyclic vs 2D cyclic.

- Choice of data distribution.
  - 1D cyclic vs 2D cyclic.
- Choice of computational maps.
  - Several options, including those for `Fan-In` and `Fan-Out`.

Implementation of `fan-both`: Design Choices

- Choice of data distribution.
  - 1D cyclic vs 2D cyclic.
- Choice of computational maps.
  - Several options, including those for `Fan-In` and `Fan-Out`.
- Communication strategy.
  - Push for data driven – send data as soon as they are available.
  - Pull for demand driven – request data when they are needed.

IMPLEMENTATION OF `fan-both`: DESIGN CHOICES

- Communication protocol.
  - 2-sided communication: send-and-receive (traditional MPI).
  - 1-sided communication: a processor puts the data directly in another processor's memory or a processor gets the data directly from another (supported by MPI-3 and GASNet).

- Communication protocol.
  - 2-sided communication: send-and-receive (traditional MPI).
  - 1-sided communication: a processor puts the data directly in another processor's memory or a processor gets the data directly from another (supported by MPI-3 and GASNet).
- Synchronous vs asynchronous communication.
  - Asynchronous communication can lead to deadlocks.

IMPLEMENTATION OF `fan-both`: DESIGN CHOICES

- Communication protocol.
  - 2-sided communication: send-and-receive (traditional MPI).
  - 1-sided communication: a processor puts the data directly in another processor's memory or a processor gets the data directly from another (supported by MPI-3 and GASNet).
- Synchronous vs asynchronous communication.
  - Asynchronous communication can lead to deadlocks.
- Scheduling of computational tasks.
  - Static or dynamic.

symPACK: a fan-both code

- symPACK is an implementation of a Fan-Both algorithm, which runs on the NERSC machines.
- The code is written in UPC++.

- The code has been tested on several matrices from the University of Florida Sparse Matrix Collection.

symPACK: A fan-both CODE

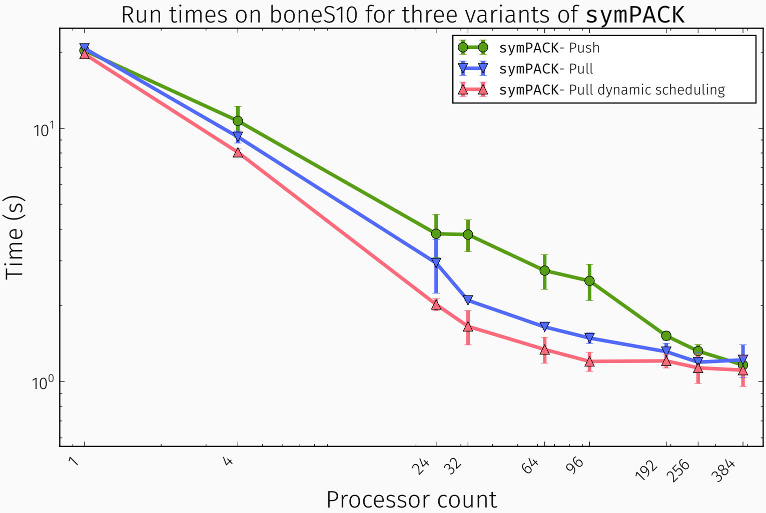
- symPACK is an implementation of a Fan-Both algorithm, which runs on the NERSC machines.
- The code is written in UPC++.
- The code has been tested on several matrices from the University of Florida Sparse Matrix Collection.
- We compared our code with the symmetric versions of MUMPS and PaStiX.

symPACK: A fan-both CODE

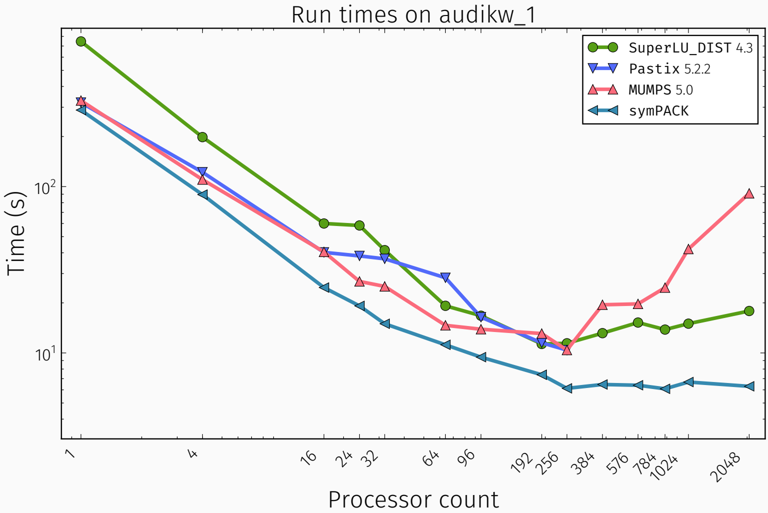
- symPACK is an implementation of a Fan-Both algorithm, which runs on the NERSC machines.
- The code is written in UPC++.

- The code has been tested on several matrices from the University of Florida Sparse Matrix Collection.

- We compared our code with the symmetric versions of MUMPS and PaStiX.
- We also compared our code with SuperLU, which is a solver for sparse nonsymmetric matrices.
  - for scaling behavior rather than actual performance ...

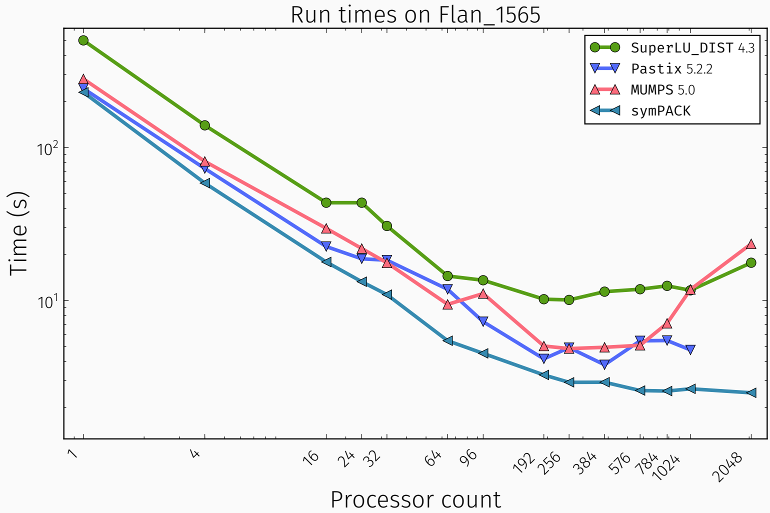


n=914,898 nnz(A)=20,896,803 nnz(L)=318,019,434

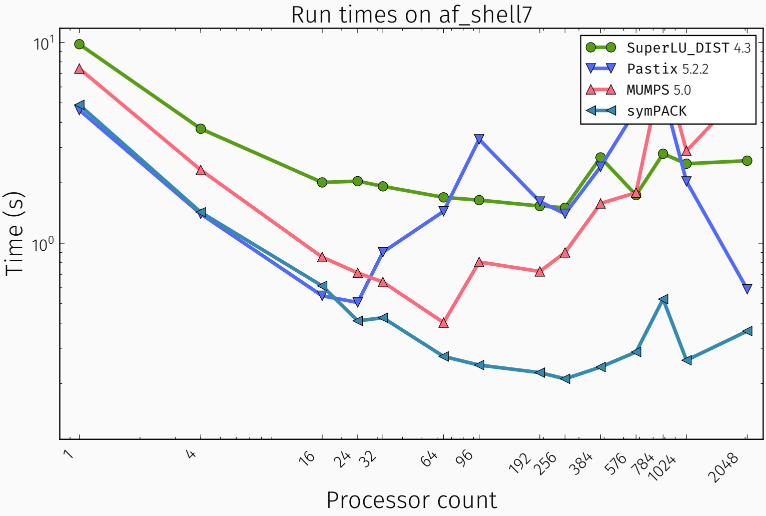


n=943,695 nnz(A)=39,297,771 nnz(L)=1,221,674,796

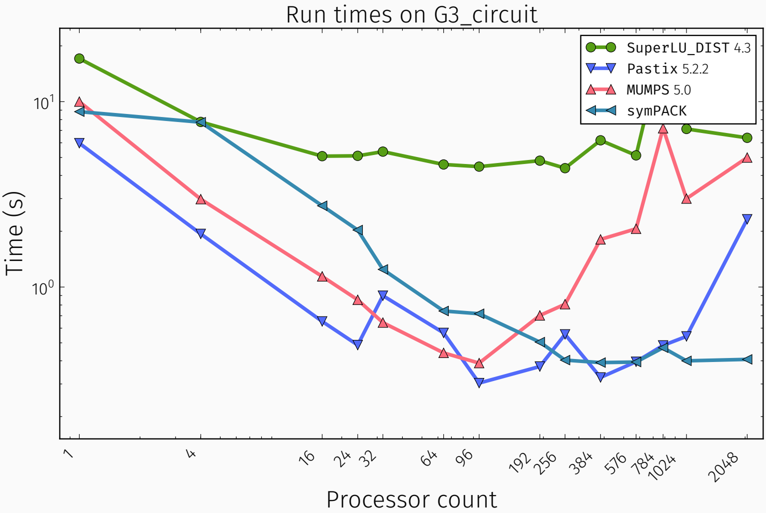
STRONG SCALING VS. STATE-OF-THE-ART



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



n=504,855 nnz(A)=17,579,155 nnz(L)=103,726,140



n=1,585,478 nnz(A)=7,660,826 nnz(L)=106,326,457

	Speedup vs. sym.			Speedup vs. best		
Problem	min	max	**avg.**	min	max	**avg.**
G3_circuit	0.24	5.70	**1.07**	0.24	5.70	**1.07**
Flan_1565	1.06	9.40	**2.11**	1.06	7.07	**1.94**
af_shell7	0.89	10.61	**3.61**	0.89	7.77	**3.21**
audikw_1	1.11	14.46	**3.14**	1.11	2.84	**1.77**
boneS10	—	—	—	0.86	4.73	**1.75**
bone010	1.06	16.83	**3.34**	1.06	2.03	**1.47**

CONCLUDING REMARKS

- We have described the `Fan-Both` algorithms.
  - Reduced communication cost in theory [Ashcraft '95].
  - Increased parallelism when performing updates.

CONCLUDING REMARKS

- We have described the `Fan-Both` algorithms.
  - Reduced communication cost in theory [Ashcraft ’95].
  - Increased parallelism when performing updates.
  - Avoiding deadlocks is challenging (similar to observation by Larkar et al.).

CONCLUDING REMARKS

- We have described the `Fan-Both` algorithms.
  - Reduced communication cost in theory [Ashcraft '95].
  - Increased parallelism when performing updates.
  - Avoiding deadlocks is challenging (similar to observation by Larkar et al.).
- New symmetric solver `symPACK`.
  - Implement `Fan-Both`.
  - Task-based Cholesky requires fine/dynamic scheduling.
  - One sided approach using UPC++.
  - Asynchronous task execution model.
  - Dynamic scheduling.

- 2D wrap mapping performance.
- Load balancing issue.

- Tree-based group communications.
- Hybrid parallelism (OpenMP).

- Data distribution (2D, block based?).
- Scheduling strategies.
- New task mapping policies.

- Investigating parallelization of the preprocessing phase (reordering and symbolic factorization).
- Pivoting for general sparse symmetric matrices.