INVENTORS FOR THE DIGITAL WORLD

Large scale SVD using polar decomposition

M. Faverge

Outline

1. Classic solution to solve large SVD problems
2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari's PhD]
3. Task based algorithms
4. Alternative solutions to (partial-SVD)

Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari's PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)

SVD - Singular Value Decomposition

$$A = U\Sigma V^T$$

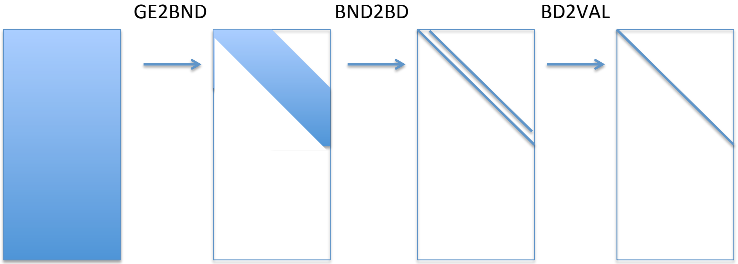
- A is general matrix
- Σ are the singular values of A
- U are the left singular vectors
- V are the right singular vectors

Singular Value Decomposition

$A = U\Sigma V^T$

- Focus on getting the singular values only (GEVAL)
- Use three steps algorithms:

GE2BND	Reduce the general matrix to general band
BND2BD	Reduce the general band to bidiagonal
BD2VAL	Compute the singular values from the bidiagonal

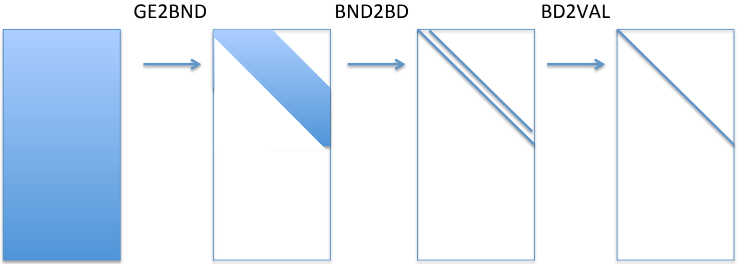


Singular Value Decomposition

$A = U\Sigma V^T$

- Focus on getting the singular values only (GEVAL)
- Use three steps algorithms:

GE2BND	**Reduce the general matrix to general band**
BND2BD	Reduce the general band to bidiagonal
BD2VAL	Compute the singular values from the bidiagonal

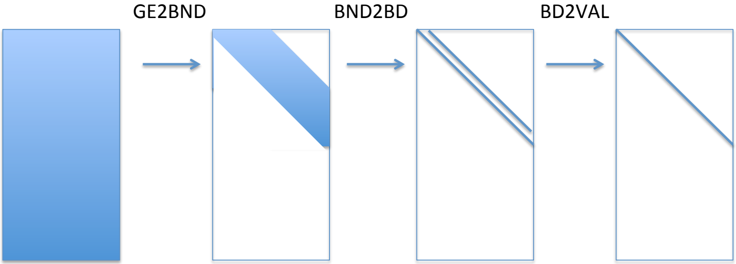


Singular Value Decomposition

$A = U\Sigma V^T$

- Focus on getting the singular values only (GEVAL)
- Use three steps algorithms:

GE2BND	**Reduce the general matrix to general band**
BND2BD	Reduce the general band to bidiagonal (PLASMA)
BD2VAL	Compute the singular values from the bidiagonal

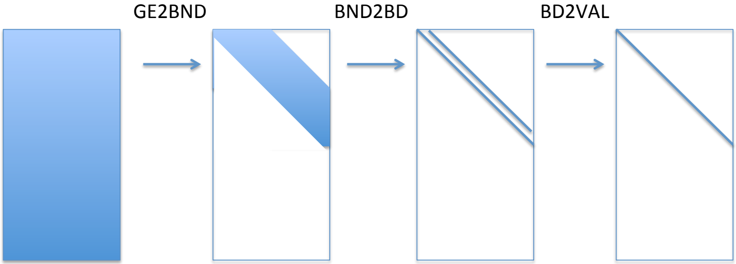


Singular Value Decomposition

$A = U\Sigma V^T$

- Focus on getting the singular values only (GEVAL)
- Use three steps algorithms:

GE2BND	**Reduce the general matrix to general band**
BND2BD	Reduce the general band to bidiagonal (PLASMA)
BD2VAL	Compute the singular values from the bidiagonal (MKL)



Two(-Three) stages algorithms

1. Reduction to tridiagonal form $A = U'BV'^T$
   - B is a bidiagonal matrix
   - U' and V' are unitary matrices
2. Find the singular values of the bidiagonal matrix B: $B = Q * \Sigma * P^t$
3. Eventually compute the eigenvectors: $U = U'Q$, and $V^t = (V'P)^t$

Problem: Reduction to tridiagonal is using BLAS 2

Two(-Three) stages algorithms

1. Reduction to tridiagonal form $A = U'BV'^T$
   - B is a bidiagonal matrix
   - U' and V' are unitary matrices
2. Find the singular values of the bidiagonal matrix B: $B = Q * \Sigma * P^t$
3. Eventually compute the eigenvectors: $U = U'Q$, and $V^t = (V'P)^t$

Problem: Reduction to tridiagonal is using BLAS 2

Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari's PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)

What is The Polar Decomposition?

- The polar decomposition:

$$\mathbf{A} = \mathbf{U_p H}, \; A \in \mathbb{R}^{m \times n} (m \geq n),$$

  where U_p is an orthogonal matrix and $H = \sqrt{A^\top A}$ is a symmetric positive semidefinite matrix
- The polar decomposition is a critical numerical algorithm for various applications, including aerospace computations, chemistry, factor analysis

A Major Building Block Toward Important DLA Algorithms

The polar decomposition can be used as a pre-processing step toward solving:

- *the symmetric eigenvalue problem*: $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top$, $V = [V_1 V_2]$
- *the singular value decomposition*: $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$

$$A = U_p H = U_p(V\Sigma V^\top) = (U_p V)\Sigma V^\top = U\Sigma V^\top$$

Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari's PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)

QDWH Polar Decomposition Algorithm

$$A = U_p H$$

where, $U_p U_p^\top = I_n$, H is symmetric positive semidefinite

- Backward stable algorithm for computing the polar decomposition
- Based on conventional computational kernels, i.e., Cholesky/QR factorizations (≤ 6 iterations for double precision) and GEMM
- The total flop count for QDWH depends on the condition number of the matrix κ:

κ	1	10^{16}
flops	$(10 + \frac{2}{3})n^3$	$43n^3$

QDWH Polar Decomposition Algorithm (cont'd)

The QDWH iteration is:

$$X_0 = A/\alpha, \begin{bmatrix} \sqrt{c_k} X_k \\ I \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R, \; X_{k+1} \quad = \frac{b_k}{c_k} X_k + \frac{1}{\sqrt{c_k}} \left(a_k - \frac{b_k}{c_k} \right) Q_1 Q_2^\top, \; k \geq 0 \tag{1}$$

When, X_k becomes well-conditioned, it is possible to replace Equation 1 with a Cholesky-based implementation as follows:

$$X_{k+1} = \frac{b_k}{c_k} X_k + \left(a_k - \frac{b_k}{c_k} \right) (X_k W_k^{-1}) W_k^{-\top}, W_k \quad = \text{chol}(Z_k), \; Z_k = I + c_k X_k^\top X_k \tag{2}$$

Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari's PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)

QDWH/ZOLO Polar Decomposition Algorithms

- QDWH:

$$\begin{bmatrix} \sqrt{c_k} X_k \\ I \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R, \; X_{k+1} = \frac{b_k}{c_k} X_k + \frac{1}{\sqrt{c_k}} \left(a_k - \frac{b_k}{c_k} \right) Q_1 Q_2^*.$$

- ZOLO:

$$\begin{bmatrix} X_k \\ \sqrt{c_{2j-1}} I \end{bmatrix} = \begin{bmatrix} Q_{j1} \\ Q_{j2} \end{bmatrix} R_j, \; X_{k+1} = X_k + \Sigma_{j=1}^{r} \frac{a_j}{\sqrt{c_{2j-1}}} Q_{j1} Q_{j2}^*.$$

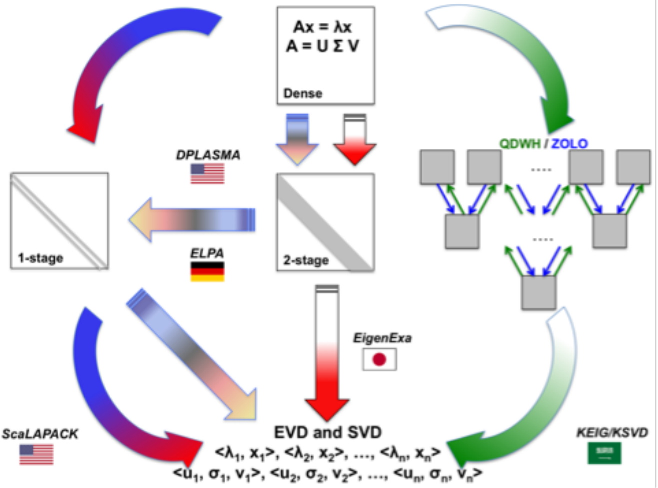
- For Ill-conditioned matrices, in double precision, QDWH converges after 6 **successive** iterations, while ZOLO converges after 2 **successive** iterations, each execute 8 **independent** embarrassingly parallel factorizations

ZOLO Arithmetic Complexity VS QDWH

Table 1: Algorithmic complexity and memory footprint for various PD algorithms with $\kappa_2(A) = 10^{12}$.

	QDWH	Successive ZOLO	Independent ZOLO
# QR-based iterations	2	8	1
# Cholesky-based iterations	4	8	1
Algorithmic complexity	$33n^3$	$100n^3$	$15\ n^3$
Memory footprint	$6n^2$	$6n^2$	$48n^2$

The Big Picture (Similar w/ SVD)



Outline

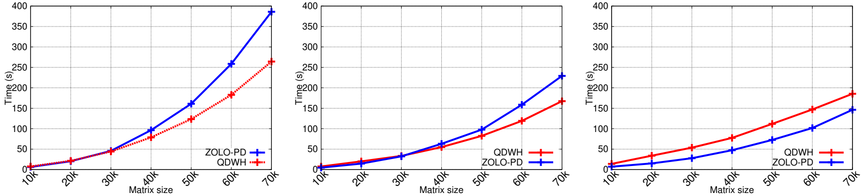
1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari's PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)

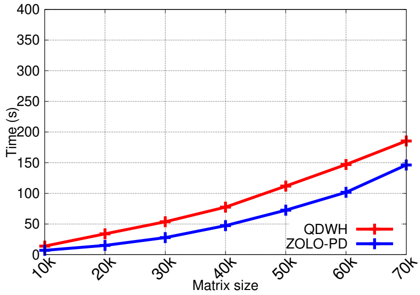
Performance Comparison on *Shaheen-2* (Polar-Decomposition)



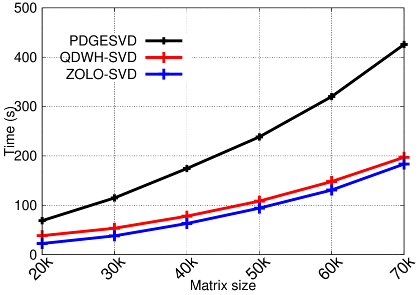
(a) 200 nodes. (b) 400 nodes. (c) 800 nodes.

Figure 1: QDWH versus ZOLO-PD.

Performance Results: From PD To SVD on 800 nodes of *Shaheen-2*



(a) Polar Decomposition



(b) SVD solvers

Outline

Summary of what is done

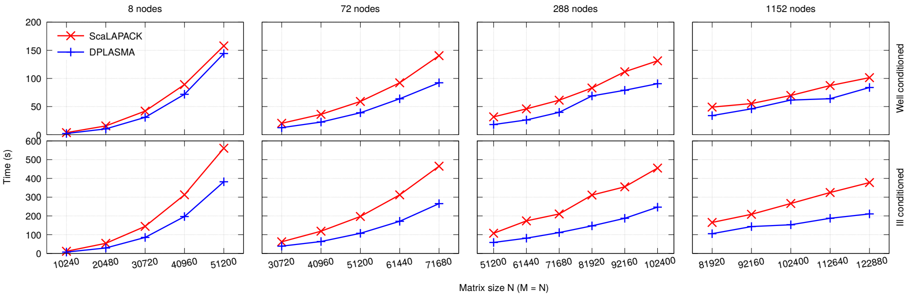
QDWH

- DPLASMA [Cluster 2019]
  Distributed memory / No GPUs
- Chameleon [TPDS 2017]
  Shared Memory / GPUs
- Distributed+GPUs ???

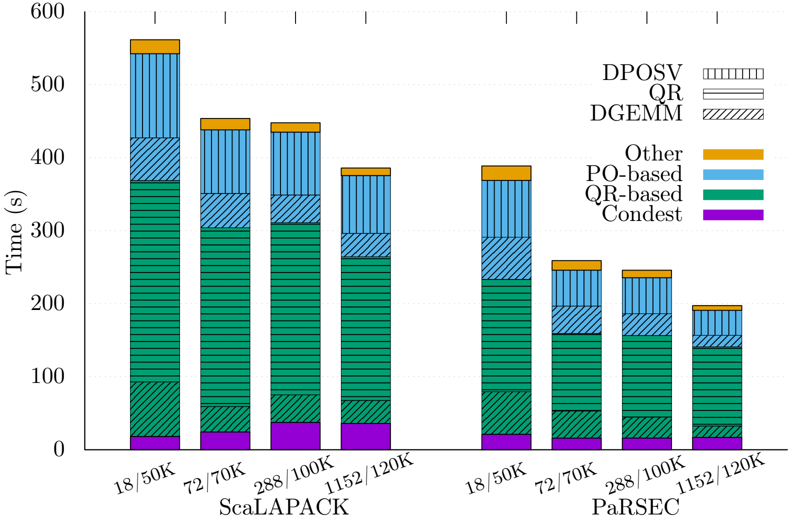
ZOLO

- Chameleon: on-going

Performance Comparisons Using Well and Ill-Conditioned Matrices



Performance Breakdown on # Nodes / Matrix Size N



What is missing for an efficient ZOLO algorithm

- Is it possible to save some memory thanks to the task-based algorithm ? (Avoid the 48 factor)
- Exploit the dynamic task-based computations to better balanced the replicated problems
- Efficient reduction step to merge the partial solutions together
- Will there be scheduling issues with the very large amount of tasks and the pipelining of the stages?

Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari's PhD]

3. Task based algorithms

4. **Alternative solutions to (partial-SVD)**

Other solutions that can be used for (partial-)SVD

- Randomized SVD (cf Diodon project)
- Partial QR with column pivoting
  Pb with the norm computations and the pivoting
- Randomized QR with column pivoting
  Similar issue as before but can localize the pivoting in a smaller matrix than can be replicated to avoid communications.
- Truncated QR factorization algorithms
  Issue with the storage of the updates
- In previous solutions, the pivoting strategy can be replace by a rotation solution, that replaces the column pivoting by a matrix-matrix product.

Thank you ☺

Sequential Task Graph: StarPU pseudo-code with Cholesky factorization

```
for (k = 0; k < NT; k+)
  potrf( RW, A[k][k] );
  for (n = k+1; n < NT; n++)
    trsm( READ, A[k][k], RW, A[k][n] );
  for (m = k+1; m < NT; m++)
    syrk( READ, A[k][m], RW, A[m][m] );
    for (n = m+1; n < NT; n++) {
      gemm( READ, A[k][m], READ, A[k][n],
                 RW, A[m][n] );
```

Leveraging QDWH-TB from Shmem to Distmem

1877 Zolotarev, best rational approximant for the scalar sign function.

1994 Higham and Papadimitriou, *SIAM, matrix inversion QDWH, shared-memory systems.*

2010 Nakatsukasa et. al, *SIAM, inverse-free QDWH, theoretical accuracy study.*

2013 Nakatsukasa and Higham, *SIAM, QDWH-EIG, QDWH-SVD, theoretical accuracy study.*

2014 Nakatsukasa, *SIAM, ZOLO-PD, ZOLO-SVD, ZOLO-EIG, theoretical accuracy study.*

2016 Sukkari, Ltaief and Keyes, *TOMS, QDWH-SVD, block algorithm, shared-memory system equipped with multiple GPUs.*

2016 Sukkari, Ltaief and Keyes, *Euro-Par, QDWH, QDWH-SVD, block algorithm, distributed-memory system.*

2017 Sukkari, Ltaief, Faverge and Keyes, TPDS, QDWH, task-based, shared-memory system equipped with multiple GPUs.

Parametrized Task Graph: PaRSEC pseudo-code with Cholesky factorization (POTRF and TRSM)

```
potrf(k)

  // Execution space
  k = 0 .. NT-1

  // Parallel partitioning
  :A(k, k)

  RW T   <- (k == 0) ? A(k, k)
              [U]
         <- (k != 0) ? T syrk(k
    -1, k) [U]

         -> T trsm(k, k+1..NT
    -1)        [U]
         -> A(k, k)
              [U]
```

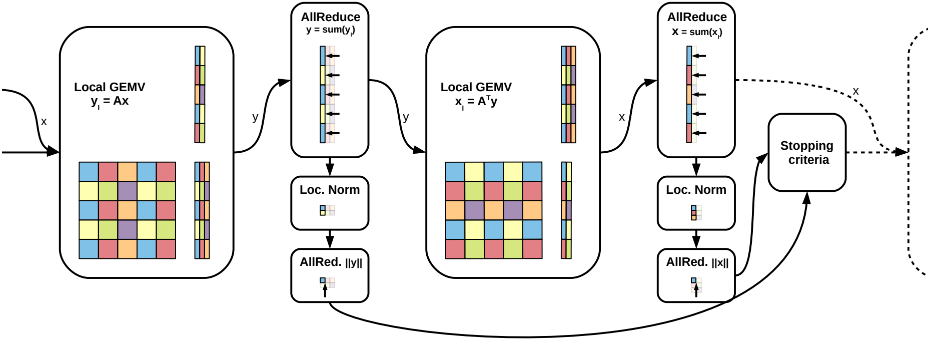
```
trsm(k, n)
  // Execution space
  k = 0   .. NT-2
  n = k+1 .. NT-1
  // Parallel partitioning
  : A(k, n)

  READ T <- T potrf(k)
              [U]
  RW   C <- (k == 0) ? A(k, n)
         <- (k != 0) ? C gemm(k
    -1, n, k)
         -> A syrk(k, n)
         -> A gemm(k, n, n+1..
    NT-1)
         -> B gemm(k, k+1..n-1,
     n)
         -> A(k, n)
```

Parametrized Task Graph: PaRSEC pseudo-code with Cholesky factorization (SYRK and GEMM)

```
syrk(k, m)
  // Execution space
  k = 0   .. NT-2
  m = k+1 .. NT-1
  // Parallel partitioning
  : A(m, m)

  READ A <- C trsm(k, m)

  RW   T <- (k == 0) ? A(m, m)
           [U]
         <- (k != 0) ? T syrk(k
    -1, m)    [U]

         -> (m == k+1) ? T
    potrf(m)       [U]
         -> (m != k+1) ? T
    syrk(k+1, m)  [U]
```

```
gemm(k, m, n)
  // Execution space
  k = 0   .. NT-3
  m = k+1 .. NT-1
  n = m+1 .. NT-1
  // Parallel partitioning
  : A(m, n)

  READ A <- C trsm(k, m)
  READ B <- C trsm(k, n)
  RW   C <- (k == 0) ? A(m, n)
         <- (k != 0) ? C gemm(k
    -1, m, n)
         -> (m == k+1) ? C
    trsm(m, n)
         -> (m != k+1) ? C
    gemm(k+1, m, n)
```

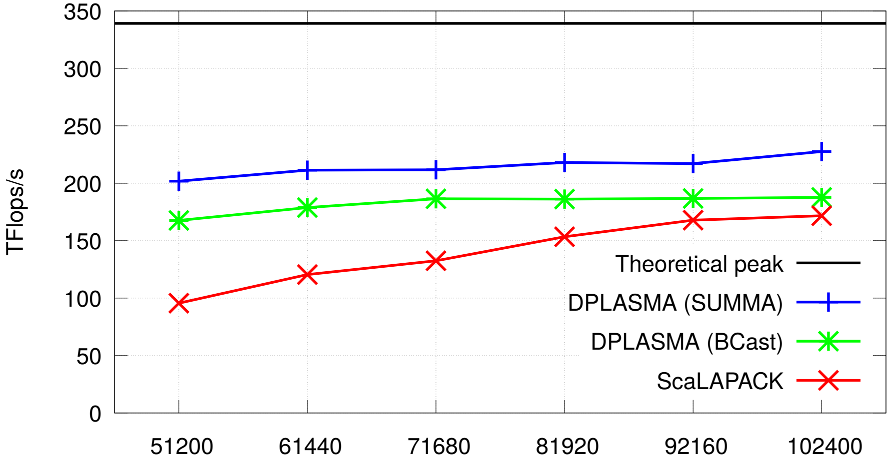
(1) Task-based Design of the Matrix Two-Norm Estimation



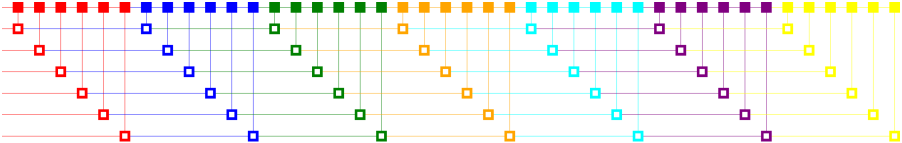
(2) Scalable Universal Matrix Multiplication Algorithm (SUMMA)

- SUMMA replaces standard broadcasts with pipelined rings of communication
- Already implemented in ScaLAPACK for distributed-memory GEMM operations
- Fine-grained computations expose a low-level control of communications, which provide more flexibility for scheduling of computational tasks and communications.
- For instance: overlapping, network congestion, communication load balancing, etc.

Performance Impact in TFlop/s on 288 nodes w/ SUMMA for Matrix-Matrix Multiplication

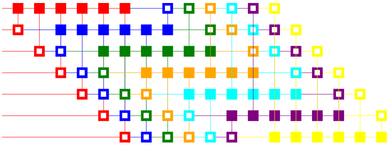


(3) Hierarchical QR Factorization Using Tree Reduction: Flat Tree $Flat(0)$ (or Domino)



Long critical path ☹
High communication volume ☹

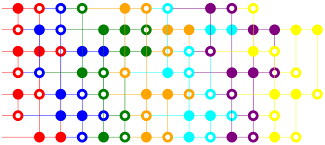
(3) Hierarchical QR Factorization Using Tree Reduction: $Flat(k)$



Short critical path ☺

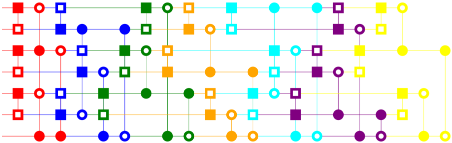
High communication volume ☹

(3) Hierarchical QR Factorization Using Tree Reduction: Greedy



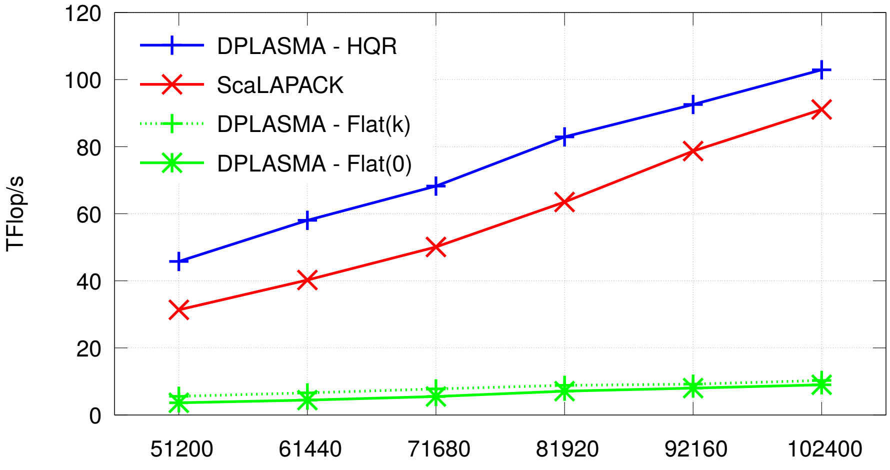
Short critical path ☺
Low communication volume ☺
Low kernels' arithmetic intensity ☹

(3) Hierarchical QR Factorization Using Tree Reduction: Mixing Greedy + flat

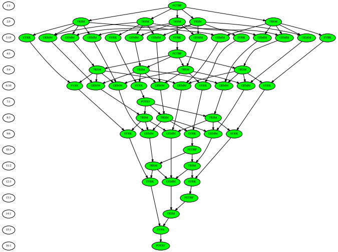


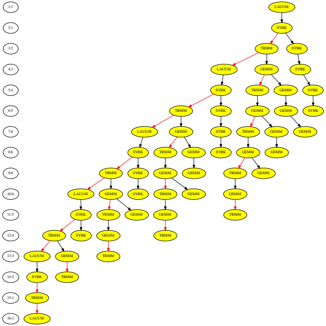
Short critical path ☺
Low communication volume ☺
High kernels' arithmetic intensity ☺

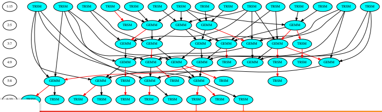
Performance Impact in TFlop/s on 288 nodes w/ HQR for the QR Factorization

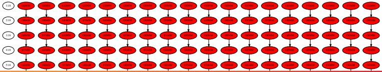


(4) Composing Directed Acyclic Graphs

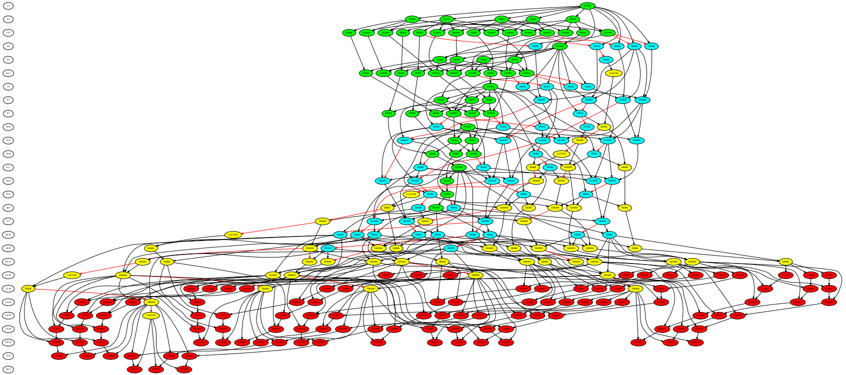








(4) Composing Directed Acyclic Graphs



Performance Impact in TFlop/s on 288 nodes w/ DAG Composition for Cholesky-based Linear Solvers