New Class of Block Matrix Orderings for the Parallel Two-Sided Jacobi SVD Algorithm

Gabriel Okša ¹, Ondrej Sýkora ², Marián Vajteršic ³

¹ Institute of Mathematics
   Slovak Academy of Sciences
   Bratislava, Slovakia

² Department of Computer Science
   Loughborough University
   Loughborough, United Kingdom

³ Institute of Scientific Computing
   University of Salzburg
   Salzburg, Austria
Outline

1. Problem formulation
2. Parallel Two-Sided Block-Jacobi Algorithm
3. Known Types of Block Ordering
4. New Clique-Based Block Ordering
5. First numerical results
Our task

Compute in parallel the Singular Value Decomposition (SVD) of a complex matrix $A$ of the size $m \times n$, $m \geq n$:

$$A = U \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} V^H,$$

where $U(m \times m)$ and $V(n \times n)$ are orthogonal and $\Sigma = \text{diag}(\sigma_i)$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$.

Numerically stable way of computation:

- one- or two-sided block-Jacobi methods;
- large degree of parallelism.

Target architecture:

- distributed memory machines (parallel supercomputers and clusters) with Message Passing Interface (MPI).
Our task

Compute in parallel the Singular Value Decomposition (SVD) of a complex matrix $A$ of the size $m \times n$, $m \geq n$:

$$A = U \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} V^H,$$

where $U(m \times m)$ and $V(n \times n)$ are orthogonal and $\Sigma = \text{diag}(\sigma_i)$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$.

Numerically stable way of computation:

- one- or two-sided block-Jacobi methods;
- large degree of parallelism.

Target architecture:

- distributed memory machines (parallel supercomputers and clusters) with Message Passing Interface (MPI).
Our task

Compute in parallel the Singular Value Decomposition (SVD) of a complex matrix $A$ of the size $m \times n$, $m \geq n$:

$$A = U \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} V^H,$$

where $U(m \times m)$ and $V(n \times n)$ are orthogonal and $\Sigma = \text{diag}(\sigma_i)$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$.

Numerically stable way of computation:

- one- or two-sided block-Jacobi methods;
- large degree of parallelism.

Target architecture:

- distributed memory machines (parallel supercomputers and clusters) with Message Passing Interface (MPI).
Main structure of algorithm

- Implemented on $p$ processors, an even blocking factor $\ell = 2p$ is used together with some ordering of subproblems.

- Each processor contains 2 block columns of $A$, $U$, $V$.

- Algorithm is based on the SVDs of $2 \times 2$-block subproblems:
  \[ S_{ij} = \begin{pmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{pmatrix}. \]

- Global termination criterion:
  \[ F(A, \ell) = \sqrt{\sum_{i,j=1, i \neq j}^{\ell} \|A_{ij}\|_F^2} < \epsilon, \quad \epsilon \equiv \text{prec} \cdot \|A\|_F. \]
Main structure of algorithm

- Implemented on $p$ processors, an even blocking factor $\ell = 2p$ is used together with some ordering of subproblems.
- Each processor contains 2 block columns of $A$, $U$, $V$.
- Algorithm is based on the SVDs of $2 \times 2$-block subproblems:

$$S_{ij} = \begin{pmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{pmatrix}.$$

- Global termination criterion:

$$F(A, \ell) = \sqrt{\sum_{i,j=1, i\neq j}^\ell \| A_{ij} \|_F^2} < \epsilon, \quad \epsilon \equiv \text{prec} \cdot \| A \|_F.$$
Main structure of algorithm

- Implemented on $p$ processors, an even blocking factor $\ell = 2p$ is used together with some ordering of subproblems.
- Each processor contains 2 block columns of $A$, $U$, $V$.
- Algorithm is based on the SVDs of $2 \times 2$-block subproblems:
  \[ S_{ij} = \begin{pmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{pmatrix}. \]

- Global termination criterion:
  \[ F(A, \ell) = \sqrt{\sum_{i,j=1, i\neq j}^{\ell} \|A_{ij}\|_F^2} < \epsilon, \quad \epsilon \equiv \text{prec} \cdot \|A\|_F. \]
Main structure of algorithm

- Implemented on $p$ processors, an even blocking factor $\ell = 2p$ is used together with some ordering of subproblems.
- Each processor contains 2 block columns of $A$, $U$, $V$.
- Algorithm is based on the SVDs of $2 \times 2$-block subproblems:
  \[ S_{ij} = \begin{pmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{pmatrix}. \]
- Global termination criterion:
  \[ F(A, \ell) = \sqrt{\sum_{i,j=1, i \neq j}^{\ell} \|A_{ij}\|^2_F} < \epsilon, \quad \epsilon \equiv \text{prec} \cdot \|A\|_F. \]
Main structure of algorithm (cont.)

- Local termination criterion:

\[ F(S_{ij}, \ell) = \sqrt{\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2} < \delta, \quad \delta \equiv \epsilon \cdot \sqrt{\frac{2}{\ell (\ell - 1)}}. \]

- Main question: How to order \(2 \times 2\)-block subproblems to be efficient?
Main structure of algorithm (cont.)

- **Local termination criterion:**

  \[ F(S_{ij}, \ell) = \sqrt{\| A_{ij} \|_F^2 + \| A_{ji} \|_F^2} < \delta, \quad \delta \equiv \epsilon \cdot \sqrt{\frac{2}{\ell (\ell - 1)}}. \]

- **Main question:** How to order 2 × 2-block subproblems to be efficient?
Cyclic block ordering

- Let the matrix $A$ be cut into $\ell \times \ell$ block structure with $\ell = 2p$ where $p$ is the number of processors.

- **Cyclic block-row ordering** define $2 \times 2$ block subproblems by block rows:
  \[(A_{12}, A_{21}), (A_{13}, A_{31}), \ldots, (A_{1\ell}, A_{\ell1}),\]
  \[(A_{23}, A_{32}), (A_{24}, A_{42}), \ldots, (A_{2\ell}, A_{\ell2}),\]
  \[\ldots, (A_{\ell-1,\ell}, A_{\ell,\ell-1}).\]

- **Cyclic bloc-column ordering** is defined similarly.

- **Parallel method**: Take $p$ consecutive pairs from the list, each pair defines one $2 \times 2$-block SVD subproblem for one processor.
Cyclic block ordering

- Let the matrix $A$ be cut into $\ell \times \ell$ block structure with $\ell = 2p$ where $p$ is the number of processors.

- **Cyclic block-row ordering** define $2 \times 2$ block subproblems by block rows:
  $$(A_{12}, A_{21}), (A_{13}, A_{31}), \ldots, (A_{1\ell}, A_{\ell 1}),$$
  $$ (A_{23}, A_{32}), (A_{24}, A_{42}), \ldots, (A_{2\ell}, A_{\ell 2}),$$
  $$ \ldots, (A_{\ell-1,\ell}, A_{\ell,\ell-1}).$$

- **Cyclic bloc-column ordering** is defined similarly.

- **Parallel method**: Take $p$ consecutive pairs from the list, each pair defines one $2 \times 2$-block SVD subproblem for one processor.
Cyclic block ordering

- Let the matrix $A$ be cut into $\ell \times \ell$ block structure with $\ell = 2p$ where $p$ is the number of processors.
- **Cyclic block-row ordering** define $2 \times 2$ block subproblems by block rows:
  
  $(A_{12}, A_{21}), (A_{13}, A_{31}), \ldots, (A_{1\ell}, A_{\ell1}),$
  
  $(A_{23}, A_{32}), (A_{24}, A_{42}), \ldots, (A_{2\ell}, A_{\ell2}),$
  
  $\ldots, (A_{\ell-1,\ell}, A_{\ell,\ell-1}).$

- **Cyclic bloc-column ordering** is defined similarly.
- **Parallel method**: Take $p$ consecutive pairs from the list, each pair defines one $2 \times 2$-block SVD subproblem for one processor.
Cyclic block ordering

- Let the matrix $A$ be cut into $\ell \times \ell$ block structure with $\ell = 2p$ where $p$ is the number of processors.
- **Cyclic block-row ordering** define $2 \times 2$ block subproblems by block rows: 
  $(A_{12}, A_{21}), (A_{13}, A_{31}), \ldots, (A_{1\ell}, A_{\ell1}), 
  (A_{23}, A_{32}), (A_{24}, A_{42}), \ldots, (A_{2\ell}, A_{\ell2}), 
  \ldots, (A_{\ell-1,\ell}, A_{\ell,\ell-1})$.
- **Cyclic bloc-column ordering** is defined similarly.
- **Parallel method**: Take $p$ consecutive pairs from the list, each pair defines one $2 \times 2$-block SVD subproblem for one processor.
Cyclic block ordering (cont.)

- During one **sweep** of the method each non-diagonal block of $A$ is nullified exactly once.

- **Main problem**: The ordering is prescribed and fixed, there is no consideration about an *actual status* (e.g., Frobenius norm) of individual blocks—may be very unefficient in decreasing the off-norm.

- Significant amount of data (e.g., whole column blocks) need to be explicitly transferred among processors at the beginning of each parallel step.
During one **sweep** of the method each non-diagonal block of $A$ is nullified exactly once.

**Main problem**: The ordering is prescribed and fixed, there is no consideration about an *actual status* (e.g., Frobenius norm) of individual blocks—may be **very unefficient** in decreasing the off-norm.

Significant amount of data (e.g., whole column blocks) need to be explicitly transferred among processors at the beginning of each parallel step.
During one **sweep** of the method each non-diagonal block of $A$ is nullified exactly once.

**Main problem:** The ordering is prescribed and fixed, there is no consideration about an *actual status* (e.g., Frobenius norm) of individual blocks—may be **very unefficient** in decreasing the off-norm.

Significant amount of data (e.g., whole column blocks) need to be explicitly transferred among processors at the beginning of each parallel step.
Dynamic Block Ordering

- **Main idea**: Take into consideration the actual status of a matrix and, in one parallel iterations step ($p$ subproblems) *decrease the off-norm as much as possible*.

- This task can be translated into the language of graph theory:
  1. Construct the weighted complete graph $G$ with $\ell$ vertices ($\ell = 2p$ is the blocking factor), where the edge $(i, j)$ has the weight $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$.
  2. At the beginning of each parallel iteration step, find the maximum-weight perfect matching on $G$, which defines the $p$ SVD subproblems—only *polynomial* complexity $O(\ell^3)$. 
Dynamic Block Ordering

- **Main idea**: Take into consideration the actual status of a matrix and, in one parallel iterations step ($p$ subproblems) *decrease the off-norm as much as possible*.
- **This task can be translated into the language of graph theory**:
  1. Construct the weighted complete graph $G$ with $\ell$ vertices ($\ell = 2p$ is the blocking factor), where the edge $(i, j)$ has the weight $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$.
  2. At the beginning of each parallel iteration step, find the maximum-weight perfect matching on $G$, which defines the $p$ SVD subproblems—only polynomial complexity $O(\ell^3)$.
Dynamic Block Ordering

- **Main idea:** Take into consideration the actual status of a matrix and, in one parallel iterations step ($p$ subproblems) *decrease the off-norm as much as possible.*

- This task can be translated into the language of graph theory:
  1. Construct the weighted complete graph $G$ with $\ell$ vertices ($\ell = 2p$ is the blocking factor), where the edge $(i, j)$ has the weight $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$.
  2. At the beginning of each parallel iteration step, find the maximum-weight perfect matching on $G$, which defines the $p$ SVD subproblems—only *polynomial* complexity $O(\ell^3)$. 
Main idea: Take into consideration the actual status of a matrix and, in one parallel iterations step ($p$ subproblems) decrease the off-norm as much as possible.

This task can be translated into the language of graph theory:

1. Construct the weighted complete graph $G$ with $\ell$ vertices ($\ell = 2p$ is the blocking factor), where the edge $(i, j)$ has the weight $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$.

2. At the beginning of each parallel iteration step, find the maximum-weight perfect matching on $G$, which defines the $p$ SVD subproblems—only polynomial complexity $O(\ell^3)$. 
Figure: Maximum-weight perfect matching for $\ell = 6$ ($p = 3$).
Clique-Based Block Ordering

- **Physical blocking factor**: for $p$ processors, let $A$ be partitioned into $p$ block columns and block rows, each processor contains one block column.

- **Logical blocking factor**: $\ell = p/r$ for some integer $r$.

- In each parallel iteration step, $\ell$ local SVDs are computed in parallel where each SVD is of block-order $r \times r$ and comprises $2r$ off-diagonal blocks of $A$.

- **Our aim**: Decrease the off-diagonal Frobenius norm in each parallel iteration step as much as possible.
Physical blocking factor: for $p$ processors, let $A$ be partitioned into $p$ block columns and block rows, each processor contains one block column.

Logical blocking factor: $\ell = p/r$ for some integer $r$.

In each parallel iteration step, $\ell$ local SVDs are computed in parallel where each SVD is of block-order $r \times r$ and comprises $2r$ off-diagonal blocks of $A$.

Our aim: Decrease the off-diagonal Frobenius norm in each parallel iteration step as much as possible.
Physical blocking factor: for \( p \) processors, let \( A \) be partitioned into \( p \) block columns and block rows, each processor contains one block column.

Logical blocking factor: \( \ell = p/r \) for some integer \( r \).

In each parallel iteration step, \( \ell \) local SVDs are computed in parallel where each SVD is of block-order \( r \times r \) and comprises \( 2r \) off-diagonal blocks of \( A \).

Our aim: Decrease the off-diagonal Frobenius norm in each parallel iteration step as much as possible.
Clique-Based Block Ordering

- **Physical blocking factor**: for $p$ processors, let $A$ be partitioned into $p$ block columns and block rows, each processor contains one block column.
- **Logical blocking factor**: $\ell = p/r$ for some integer $r$.
- In each parallel iteration step, $\ell$ local SVDs are computed in parallel where each SVD is of block-order $r \times r$ and comprises $2r$ off-diagonal blocks of $A$.
- **Our aim**: Decrease the off-diagonal Frobenius norm in each parallel iteration step as much as possible.
Graph-Theoretical Formulation

Let $G$ be a complete graph with $p$ vertices (= physical blocking factor). Let the edge $(i, j)$ be weighted by $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$.

**Task:** Partition $G$ into $\ell/2$ disjoint cliques of size $r$ where the weight of cliques (i.e., the sum of weights through the edges belonging to chosen cliques) is maximized.

For $r = 2$ this is just the maximum-weight perfect matching used in the Dynamic Ordering.

**Main problem:** For $r \geq 3$, this task is NP-hard.
Graph-Theoretical Formulation

Let $G$ be a complete graph with $p$ vertices ($=$ physical blocking factor). Let the edge $(i, j)$ be weighted by $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$.

**Task:** Partition $G$ into $\ell/2$ disjoint cliques of size $r$ where the weight of cliques (i.e., the sum of weights through the edges belonging to chosen cliques) is maximized.

For $r = 2$ this is just the maximum-weight perfect matching used in the Dynamic Ordering.

**Main problem:** For $r \geq 3$, this task is NP-hard.
Let $G$ be a complete graph with $p$ vertices (＝ physical blocking factor). Let the edge $(i, j)$ be weighted by $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$.

**Task**: Partition $G$ into $\ell/2$ disjoint cliques of size $r$ where the weight of cliques (i.e., the sum of weights through the edges belonging to chosen cliques) is maximized.

For $r = 2$ this is just the maximum-weight perfect matching used in the Dynamic Ordering.

**Main problem**: For $r \geq 3$, this task is NP-hard.
Graph-Theoretical Formulation

Let $G$ be a complete graph with $p$ vertices (= physical blocking factor). Let the edge $(i, j)$ be weighted by $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$.

**Task**: Partition $G$ into $\ell/2$ disjoint cliques of size $r$ where the weight of cliques (i.e., the sum of weights through the edges belonging to chosen cliques) is maximized.

For $r = 2$ this is just the maximum-weight perfect matching used in the Dynamic Ordering.

**Main problem**: For $r \geq 3$, this task is NP-hard.
CLIQUE 1: INDICES (1, 3, 5) (blocks and processors)
CLIQUE 2: INDICES (2, 4, 6) (blocks and processors)

Figure: Clique-based block ordering for $p = 6$, $r = 3$. 
Example (cont.)

- In the example above there are 2 local SVDs defined by submatrices with block indices (1, 3, 5) and (2, 4, 6):

  \[
  \text{CL1} = \begin{pmatrix}
  A_{11} & A_{13} & A_{15} \\
  A_{31} & A_{33} & A_{35} \\
  A_{51} & A_{53} & A_{55}
\end{pmatrix}, \quad \text{CL2} = \begin{pmatrix}
  A_{22} & A_{24} & A_{26} \\
  A_{42} & A_{44} & A_{46} \\
  A_{62} & A_{64} & A_{66}
\end{pmatrix}.
  \]

- MPI library: The SVD of CL1 is computed in the context that consists of processors 1, 3 and 5. Similarly, the SVD of CL2 is computed in the context that consists of processors 2, 4 and 6.

- In general, there are \( \ell \) contexts, each with \( r \) processors.

- Important: The matrix data is not moved explicitly between processors at the beginning of a parallel iteration step—just new contexts are created.
Example (cont.)

- In the example above there are 2 local SVDs defined by submatrices with block indices \((1, 3, 5)\) and \((2, 4, 6)\):

\[
\text{CL1} = \begin{pmatrix}
A_{11} & A_{13} & A_{15} \\
A_{31} & A_{33} & A_{35} \\
A_{51} & A_{53} & A_{55}
\end{pmatrix}, \quad \text{CL2} = \begin{pmatrix}
A_{22} & A_{24} & A_{26} \\
A_{42} & A_{44} & A_{46} \\
A_{62} & A_{64} & A_{66}
\end{pmatrix}.
\]

- **MPI library**: The SVD of \(\text{CL1}\) is computed in the context that consists of processors 1, 3 and 5. Similarly, the SVD of \(\text{CL2}\) is computed in the context that consists of processors 2, 4 and 6.

- In general, there are \(\ell\) contexts, each with \(r\) processors.

- **Important**: The matrix data is not moved explicitly between processors at the beginning of a parallel iteration step—just new contexts are created.
Example (cont.)

- In the example above there are 2 local SVDs defined by submatrices with block indices (1, 3, 5) and (2, 4, 6):

\[
\text{CL}_1 = \begin{pmatrix} A_{11} & A_{13} & A_{15} \\ A_{31} & A_{33} & A_{35} \\ A_{51} & A_{53} & A_{55} \end{pmatrix}, \quad \text{CL}_2 = \begin{pmatrix} A_{22} & A_{24} & A_{26} \\ A_{42} & A_{44} & A_{46} \\ A_{62} & A_{64} & A_{66} \end{pmatrix}.
\]

- **MPI library**: The SVD of \( \text{CL}_1 \) is computed in the context that consists of processors 1, 3 and 5. Similarly, the SVD of \( \text{CL}_2 \) is computed in the context that consists of processors 2, 4 and 6.

- In general, there are \( \ell \) contexts, each with \( r \) processors.

- **Important**: The matrix data is not moved explicitly between processors at the beginning of a parallel iteration step—just new contexts are created.
Example (cont.)

- In the example above there are 2 local SVDs defined by submatrices with block indices (1, 3, 5) and (2, 4, 6):

\[
\text{CL1} = \begin{pmatrix} A_{11} & A_{13} & A_{15} \\ A_{31} & A_{33} & A_{35} \\ A_{51} & A_{53} & A_{55} \end{pmatrix}, \quad \text{CL2} = \begin{pmatrix} A_{22} & A_{24} & A_{26} \\ A_{42} & A_{44} & A_{46} \\ A_{62} & A_{64} & A_{66} \end{pmatrix}.
\]

- **MPI library**: The SVD of CL1 is computed in the context that consists of processors 1, 3 and 5. Similarly, the SVD of CL2 is computed in the context that consists of processors 2, 4 and 6.

- In general, there are \( \ell \) contexts, each with \( r \) processors.

- **Important**: The matrix data is not moved explicitly between processors at the beginning of a parallel iteration step—just new contexts are created.
Work in one parallel iteration step

1. Given $p$, $r$, $\ell$ and actual weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$, find $\ell$ maximum-weight disjoint cliques using a genetic serial algorithm in processor 1—this defines the new block ordering.

2. Broadcast the new block ordering to all processors.

3. Delete old contexts.

4. Create $\ell$ contexts based on the new ordering, each context of size $r$, and compute $\ell$ SVDs of block size $r \times r$ in parallel by the two-sided Jacobi method.

5. Update left and right singular vectors in all processors by matrix multiplications.

6. Update weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$. 
1. Given $p$, $r$, $\ell$ and actual weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$, find $\ell$ maximum-weight disjoint cliques using a genetic serial algorithm in processor 1—this defines the new block ordering.

2. Broadcast the new block ordering to all processors.

3. Delete old contexts.

4. Create $\ell$ contexts based on the new ordering, each context of size $r$, and compute $\ell$ SVDs of block size $r \times r$ in parallel by the two-sided Jacobi method.

5. Update left and right singular vectors in all processors by matrix multiplications.

6. Update weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$. 
Work in one parallel iteration step

1. Given $p$, $r$, $\ell$ and actual weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$, find $\ell$ maximum-weight disjoint cliques using a genetic serial algorithm in processor 1—this defines the new block ordering.

2. Broadcast the new block ordering to all processors.

3. Delete old contexts.

4. Create $\ell$ contexts based on the new ordering, each context of size $r$, and compute $\ell$ SVDs of block size $r \times r$ in parallel by the two-sided Jacobi method.

5. Update left and right singular vectors in all processors by matrix multiplications.

6. Update weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$. 
Work in one parallel iteration step

1. Given $p$, $r$, $\ell$ and actual weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$, find $\ell$ maximum-weight disjoint cliques using a genetic serial algorithm in processor 1—this defines the new block ordering.

2. Broadcast the new block ordering to all processors.

3. Delete old contexts.

4. Create $\ell$ contexts based on the new ordering, each context of size $r$, and compute $\ell$ SVDs of block size $r \times r$ in parallel by the two-sided Jacobi method.

5. Update left and right singular vectors in all processors by matrix multiplications.

6. Update weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$. 
Work in one parallel iteration step

1. Given $p$, $r$, $\ell$ and actual weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$, find $\ell$ maximum-weight disjoint cliques using a genetic serial algorithm in processor 1—this defines the new block ordering.

2. Broadcast the new block ordering to all processors.

3. Delete old contexts.

4. Create $\ell$ contexts based on the new ordering, each context of size $r$, and compute $\ell$ SVDs of block size $r \times r$ in parallel by the two-sided Jacobi method.

5. Update left and right singular vectors in all processors by matrix multiplications.

6. Update weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$. 
Work in one parallel iteration step

1. Given $p$, $r$, $\ell$ and actual weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$, find $\ell$ maximum-weight disjoint cliques using a genetic serial algorithm in processor 1—this defines the new block ordering.

2. Broadcast the new block ordering to all processors.

3. Delete old contexts.

4. Create $\ell$ contexts based on the new ordering, each context of size $r$, and compute $\ell$ SVDs of block size $r \times r$ in parallel by the two-sided Jacobi method.

5. Update left and right singular vectors in all processors by matrix multiplications.

6. Update weights $\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2$. 
Genetic algorithm

- The serial genetic algorithm for the maximum-weight clique partition has been implemented in C++ using the free available library GALIB (v.2.4.6) from MIT, USA.
- The genome is the 2D binary string of dimensions $\ell \times p$ that represents one partition into disjoint cliques.
- The crossover function creates two new genomes (son and daughter) from two old genomes (parents) by choosing the heavier individual cliques (better parent is repeated).
- The objective function, which should be maximized, is the weight of a genome.
- Typical run:
  population size = 20,
  number of generations = $10^4$. 
The serial genetic algorithm for the maximum-weight clique partition has been implemented in C++ using the free available library GALIB (v.2.4.6) from MIT, USA.

The genome is the 2D binary string of dimensions $\ell \times p$ that represents one partition into disjoint cliques.

The crossover function creates two new genomes (son and daughter) from two old genomes (parents) by choosing the heavier individual cliques (better parent is repeated).

The objective function, which should be maximized, is the weight of a genome.

Typical run:
- population size = 20,
- number of generations = $10^4$. 
The serial genetic algorithm for the maximum-weight clique partition has been implemented in C++ using the free available library GALIB (v.2.4.6) from MIT, USA.

- The **genome** is the 2D binary string of dimensions $\ell \times p$ that represents one partition into disjoint cliques.
- The **crossover** function creates two new genomes (son and daughter) from two old genomes (parents) by choosing the heavier individual cliques (better parent is repeated).

- The **objective** function, which should be maximized, is the weight of a genome.

**Typical run:**
- population size = 20,
- number of generations = $10^4$. 
Genetic algorithm

- The serial genetic algorithm for the maximum-weight clique partition has been implemented in C++ using the free available library GALIB (v.2.4.6) from MIT, USA.
- The **genome** is the 2D binary string of dimensions $\ell \times p$ that represents one partition into disjoint cliques.
- The **crossover** function creates two new genomes (son and daughter) from two old genomes (parents) by choosing the heavier individual cliques (better parent is repeated).
- The **objective** function, which should be maximized, is the weight of a genome.

Typical run:

- population size = 20,
- number of generations = $10^4$. 

Genetic algorithm

- The serial genetic algorithm for the maximum-weight clique partition has been implemented in C++ using the free available library GALIB (v.2.4.6) from MIT, USA.
- The genome is the 2D binary string of dimensions $\ell \times p$ that represents one partition into disjoint cliques.
- The crossover function creates two new genomes (son and daughter) from two old genomes (parents) by choosing the heavier individual cliques (better parent is repeated).
- The objective function, which should be maximized, is the weight of a genome.
- Typical run:
  - population size = 20,
  - number of generations = $10^4$. 
**Table:** Performance for $p = 12$, $prec = 10^{-13}$, $\kappa = 10$, multiple minimal SV. $T_p$ is in seconds, $R_{GA} = (T_{GA}/T_p) \times 100$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>2 cliques</th>
<th></th>
<th>6 cliques</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n_{iter}$</td>
<td>$T_p$</td>
<td>$R_{GA}$</td>
<td>$n_{iter}$</td>
</tr>
<tr>
<td>1000</td>
<td>37</td>
<td>98.8</td>
<td>4.3</td>
<td>409</td>
</tr>
<tr>
<td>2000</td>
<td>39</td>
<td>463.5</td>
<td>1.0</td>
<td>416</td>
</tr>
<tr>
<td>3000</td>
<td>38</td>
<td>1393.3</td>
<td>0.3</td>
<td>406</td>
</tr>
<tr>
<td>4000</td>
<td>36</td>
<td>3084.8</td>
<td>0.1</td>
<td>402</td>
</tr>
<tr>
<td>5000</td>
<td>37</td>
<td>6144.0</td>
<td>0.1</td>
<td>403</td>
</tr>
<tr>
<td>6000</td>
<td>37</td>
<td>9994.2</td>
<td>&lt; 0.1</td>
<td>427</td>
</tr>
<tr>
<td>7000</td>
<td>37</td>
<td>15926.5</td>
<td>&lt; 0.1</td>
<td>412</td>
</tr>
<tr>
<td>8000</td>
<td>35</td>
<td>23757.1</td>
<td>&lt; 0.1</td>
<td>423</td>
</tr>
<tr>
<td>9000</td>
<td>37</td>
<td>34040.0</td>
<td>&lt; 0.1</td>
<td>417</td>
</tr>
<tr>
<td>10000</td>
<td>38</td>
<td>56532.5</td>
<td>&lt; 0.1</td>
<td>431</td>
</tr>
</tbody>
</table>