Algorithms for Dense Matrices I

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 14/15

Topics

Algorithms for dense matrices as data parallel algorithms

- Data distribution of vectors and matrices
- Matrix transposition

Partitioning of Vectors I

- Vector $x \in \mathcal{R}^N$ corresponds to an ordered list of numbers.
- Each index i of the index set $I = \{0, \ldots, N-1\}$ is assigned a real number x_i.
- Instead of $\mathcal{R}^N$ we write $\mathcal{R}(I)$ to emphasize the dependency of the index set.
- The natural (and most efficient) data structure for a vector is the array.
- Since arrays start in many programming languages with index 0, this is also the case for the index set I.

Partitioning of Vectors II

- A data partitioning matches now a segmentation of the index set I

$$I = \bigcup_{p \in P} I_p, \text{ with } p \neq q \Rightarrow I_p \cap I_q = \emptyset,$$

  where P is the process set.
- With good load balancing the index sets I_p, $p \in P$ should contain each (nearly) an equal number of elements.
- Process $p \in P$ stores such the components x_i, $i \in I_p$ of the vector x.
- In each process we would again like to work with a contiguous index set $\tilde{I}_p$, that starts at 0, this means

$$\tilde{I}_p = \{0, \ldots, |I_p| - 1\}.$$

Partitioning of Vectors III

The mappings

$$\begin{aligned} p\colon \quad & I \to P \text{ resp.} \\ \mu\colon \quad & I \to \mathbf{N} \end{aligned}$$

assign each index $i \in I$ invertible unique a process $p(i) \in P$ and a local index $\mu(i) \in \tilde{I}_{p(i)}$:

$$I \ni i \mapsto (p(i), \mu(i)).$$

The invertible mapping

$$\mu^{-1}\colon \underbrace{\bigcup_{p \in P} \{p\} \times \tilde{I}_p}_{\subset P \times \mathbf{N}} \to I$$

provides for each local index $i \in \tilde{I}_p$ and process $p \in P$ the global index $\mu^{-1}(p, i)$, thus

$$p(\mu^{-1}(p, i)) = p \text{ and } \mu(\mu^{-1}(p, i)) = i.$$

Partitioning of Vectors IV

Common partitionings are especially the *cyclic partitioning* with[1]

$$\begin{aligned} p(i) &= i \,\%\, P \\ \mu(i) &= i \div P \end{aligned}$$

and the *blockwise partitioning* with

$$\begin{aligned} p(i) &= \begin{cases} i \div (B+1) & \text{if } i < R(B+1) \\ R + (i - R(B+1)) \div B & \text{otherwise} \end{cases} \\ \mu(i) &= \begin{cases} i \,\%(B+1) & \text{if } i < R(B+1) \\ (i - R(B+1)) \,\%\, B & \text{otherwise} \end{cases} \end{aligned}$$

with $B = N \div P$ and $R = N \,\%\, P$. Here is the idea, that the first R processes get $B+1$ indices and the remaining B indices each.

[1] $\div$ means integer division; % the modulo function

Partitioning of Vectors V

Cyclic and blockwise partitioning for $N = 13$ and $P = 4$:
cyclic partitioning:

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	1	2	3	0	1	2	3	0	1	2	3	0
$\mu(i)$:	0	0	0	0	1	1	1	1	2	2	2	2	3

$$\text{z.B. } I_1 = \{1, 5, 9\}, \quad \tilde{I}_1 = \{0, 1, 2\}.$$

blockwise partitioning

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	0	0	0	1	1	1	2	2	2	3	3	3
$\mu(i)$:	0	1	2	3	0	1	2	0	1	2	0	1	2

$$\text{z.B. } I_1 = \{4, 5, 6\}, \quad \tilde{I}_1 = \{0, 1, 2\}.$$

Partitioning of Matrices I

- For a matrix $A \in \mathcal{R}^{N \times M}$ each tuple $(i, j) \in I \times J$, with $I = \{0, \dots, N-1\}$ and $J = \{0, \dots, M-1\}$, is assigned a real number a_{ij}.
- In principle the assignment of matrix elements to processors is arbitrary
- However the elements assigned to a processor can in general *not* be represented as matrix again.
- Exception: separate segmentation of the one-dimensional index sets I and J.
- Herefore we assume the processes as being organized as a two-dimensional field , thus

$$(p, q) \in \{0, \dots, P-1\} \times \{0, \dots, Q-1\}.$$

Partitioning of Matrices II

- The index sets I, J are partitioned into

$$I = \bigcup_{p=0}^{P-1} I_p \text{ and } J = \bigcup_{q=0}^{Q-1} J_q$$

- process (p, q) is then responsible for the indices $I_p \times J_q$.
- Locally process (p, q) stores its elements then as $\mathcal{R}(\tilde{I}_p \times \tilde{J}_q)$ matrix.
- The partitioning of I and J are formally described by the mappings p and μ as well as q and ν:

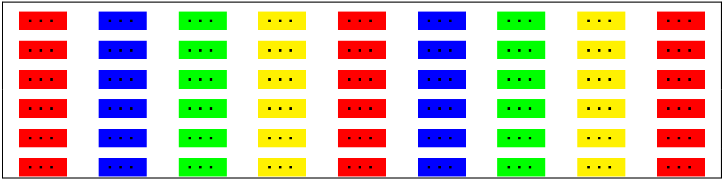
$$\begin{aligned} I_p &= \{i \in I \mid p(i) = p\}, \quad & \tilde{I}_p &= \{n \in \mathbf{N} \mid \exists i \in I : p(i) = p \wedge \mu(i) = n\} \\ J_q &= \{j \in J \mid q(j) = q\}, \quad & \tilde{J}_q &= \{m \in \mathbf{N} \mid \exists j \in J : q(j) = q \wedge \nu(j) = m\} \end{aligned}$$

Partitioning of Matrices III

Examples for partitioning of a 6×9 matrix onto four processors

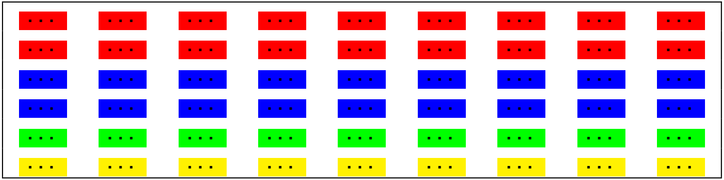
(a) $P = 1$, $Q = 4$ (Columns), J: cyclic:

0	1	2	3	4	5	6	7	8	J
0	1	2	3	0	1	2	3	0	q
0	0	0	0	1	1	1	1	2	ν



(b) $P = 4, Q = 1$ (Rows), I: blockwise:

0	0	0
1	0	1
2	1	0
3	1	1
4	2	0
5	3	0
I	p	μ



Partitioning of Matrices IV

(c) $P = 2$, $Q = 2$ (Array), I: cyclic, J: blockwise:

			0	1	2	3	4	5	6	7	8	J
			0	0	0	0	0	1	1	1	1	q
			0	1	2	3	4	0	1	2	3	ν
0	0	0	…	…	…	…	…	…	…	…	…	
1	1	0	…	…	…	…	…	…	…	…	…	
2	0	1	…	…	…	…	…	…	…	…	…	
3	1	1	…	…	…	…	…	…	…	…	…	
4	0	2	…	…	…	…	…	…	…	…	…	
5	1	2	…	…	…	…	…	…	…	…	…	
I	p	μ										

Partitioning of Matrices V

Which data partitioning is now the best one?

- In general the organisation of the processes as a nearly quadratic array leads to a partitioning with good load balancing.
- More important is however that different partitionings are suited differently good for distinct algorithms.
- We will see, that a process array with cyclic partitioning is suited quite well for row as well as column indices for the *LU* partitioning.
- This partitioning is however not optimal for the solution of the resulting triangular systems. If one has to solve the equation system for many righthand sides then a compromise has to be achieved.
- This generally holds for nearly all tasks of linear algebra: The multiplication of two matrices or the transposition of a matrix represents only a step in a larger algorithm.
- The data partitioning can thus not be optimized towards a partial step, but should give a meaningful tradeoff. Eventually can be thought whether rearranging (copying) the data into a different structure is advantegeous.

Transposition of a Matrix I

Task description
Given: $A \in \mathcal{R}^{N \times M}$ distributed onto a set of processes;
Determine: A^T with the same data partitioning as A.

- In principle the problem is trivial.
- We could distribute the matrix onto the processors such, that only communication with nearest neighbors is necessary (since the processes communicate pairwise).

12	1	3	5
0	13	7	9
2	6	14	11
4	8	10	15

Optimal data distribution for the matrix transposition (the numbers denote the processor numbers).

Transposition of a Matrix II

Example with ring topology:

- Obviously only communication is necessary between direct neighbors ($0 \leftrightarrow 1, 2 \leftrightarrow 3, \ldots, 10 \leftrightarrow 11$).
- Albeit these data partitioning does not coincide with the scheme, that we just have introduced and is for example less suited for the multiplication of two matrices.

Transposition of a Matrix: 1D Partitioning

Let us consider without loss of generality a column-wise, blocked partitioning

$$
\begin{array}{r|c|c|c|}
 & \overbrace{\hphantom{(0,0)\quad(0,1)\quad\cdots}}^{N/P} & & \\
\hline
 & \begin{matrix}(0,0) & (0,1) & \cdots \\ (1,0) & (1,1) & \\ \vdots & & \ddots\end{matrix} & \begin{matrix}\cdots & & \cdots \\ & \text{to } P_0 & \\ & & \end{matrix} & \begin{matrix} & (0,7) \\ \text{to } P_0 & \\ & \end{matrix} \\
\hline
N/P\left\{\vphantom{\begin{matrix}a\\a\\a\\a\end{matrix}}\right. & \begin{matrix}\vdots & \\ & \text{to } P_1 \\ \vdots & \end{matrix} & \ddots & \text{to } P_1 \\
\hline
 & \begin{matrix}\vdots & \\ (7,0) & \text{to } P_2\end{matrix} & \begin{matrix} \\ \text{to } P_2\end{matrix} & \begin{matrix}\ddots & \\ & (7,7)\end{matrix} \\
\hline
 & P_0 & P_1 & P_2
\end{array}
$$

8×8 matrix on three processors in column-wise, blocked distribution.

Transposition of a Matrix: 1D Partitioning

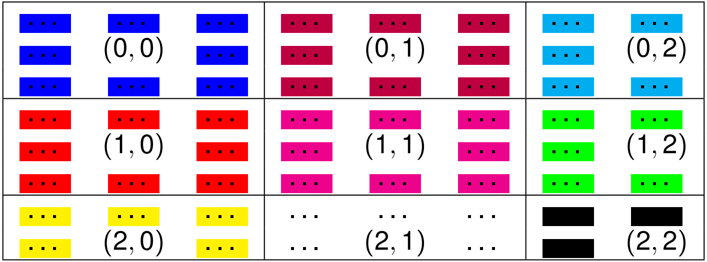
- Obviously in this case each processor has to send data to each other.
- Thus an all-to-all communication with individual messages has to be performed.
- Let us assume a hypercube structure as connection topology, then we get the following parallel runtime for a $N \times N$ matrix and P processors:

$$
\begin{aligned}
T_P(N,P) &= \underbrace{2(t_s + t_h)\operatorname{ld} P}_{\text{setup}} + \underbrace{t_w \frac{N^2}{P^2} P \operatorname{ld} P}_{\text{data trans-mission}} + \underbrace{(P-1)\frac{N^2}{P^2}\frac{t_e}{2}}_{\text{transposition}} \approx \\
&\approx \operatorname{ld} P (t_s + t_h) 2 + \frac{N^2}{P} \operatorname{ld} P t_w + \frac{N^2}{P}\frac{t_e}{2}
\end{aligned}
$$

- Also for fixed P and increasing N we cannot make the communication share of the total runtime arbitrary small.
- This is the same for all algorithms for transposition (also for an optimal distribution as above).
- Matrix transposition has therefore no iso-efficiency function and is not scalable.

Transposition of a Matrix: 2D Partitioning

We consider now the two-dimensional, blocked distribution of a $N \times N$ matrix onto a $\sqrt{P} \times \sqrt{P}$ processor array:

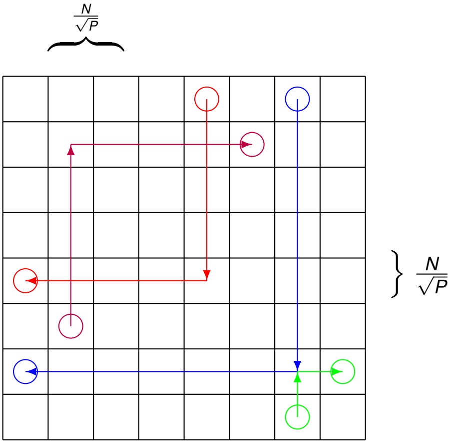


Example for a two-dimensional, blocked distribution $N = 8$, $\sqrt{P} = 3$.

Transposition of a Matrix

- Each processor has to exchange its partial matrix with exactly one other.
- A naive transposition algorithm for these configuration is:
  - Processors (p, q) below the main diagonal ($p > q$) send the partial matrix in the column to above up to processor (q, q), thereafter the partial matrix is routed to the right up to the final column to processor (q, p).
  - Corresponding the data of processors (p, q) are routed above the main diagonal ($q > p$) first in the column q to below up to (q, q) and then to the left until (q, p) is reached.

Transposition of a Matrix



Diverse pathes of partial matrices for $\sqrt{P} = 8$.

Transposition of a Matrix

- Obviously route the processors (p, q) with $p > q$ data from below to above resp. right to left and processors (p, q) with $q > p$ correspondingly data from above to below and left to right.
- For synchronous communication in each step four send- resp. receive operations are necessary, and in total one needs $2(\sqrt{P} - 1)$ steps.
- The parallel runtime therefore amounts

$$
\begin{aligned}
T_P(N, P) &= 2(\sqrt{P} - 1) \cdot 4 \left(t_s + t_h + t_w \left(\frac{N}{\sqrt{P}} \right)^2 \right) + \frac{1}{2} \left(\frac{N}{\sqrt{P}} \right)^2 t_e \approx \\
&\approx \sqrt{P} 8 (t_s + t_h) + \frac{N^2}{P} \sqrt{P} 8 t_w + \frac{N^2}{P} \frac{t_e}{2}
\end{aligned}
$$

- In comparison to a one-dimensional distribution with hypercube one has in the data transmission the factor $\sqrt{P}$ instead of ld P.

Recursive Transposition Algorithm

This algorithm is based on the following observation: For a 2×2 block matrix partitioning of A applies

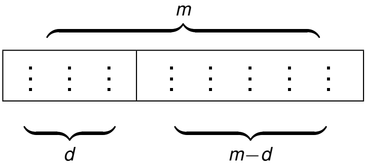
$$A^T = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}^T = \begin{pmatrix} A_{00}^T & A_{10}^T \\ A_{01}^T & A_{11}^T \end{pmatrix}$$

thus the off-diagonal blocks change the places and then each partial matrix has to be transposed. This of course happens recursively until a 1×1 matrix is reached. Is $N = 2^n$, then n recursion steps are necessary.

Recursive Transposition Algorithm

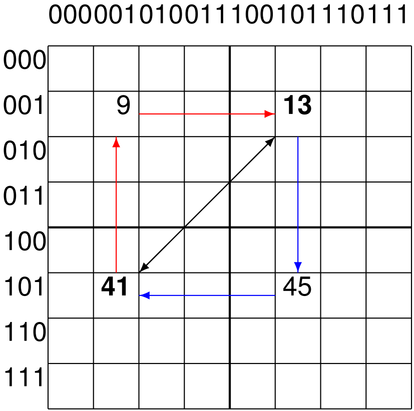
- The hypercube is the ideal connection topology for this algorithm.
- With $N = 2^n$ and $\sqrt{P} = 2^d$ with $n \geq d$ this mapping of indices $I = \{0, \ldots, N-1\}$ is done on the processors via

$$\begin{aligned} p(i) &= i \div 2^{m-d}, \\ \mu(i) &= i \,\%\, 2^{m-d} \end{aligned}$$



- The upper d bits of an index describe the processor, on which the index is mappeed.
- Consider as example $d = 3$, thus $\sqrt{P} = 2^3 = 8$.
- In the recursion step the matrix has to be divided into 2×2 blocks from 4×4 partial matrices and $2 \cdot 16$ processors have to exchange data, for example processor $101001 = 41$ and $001101 = 13$. This happens in two steps over the processors $001001 = 9$ and $101101 = 45$.
- These are both *direct* neighbors of the processors 41 and 13 in the hypercube.

Recursive Transposition Algorithm



Communication in the recursive transposition algorithm for $d = 3$.

The recursive transposition algorithm works now recursive on the processor topology. Is *a* processor reached, the transposition is continued with the sequential algorithms. The parallel runtime is described with

$$T_P(N, P) = \operatorname{ld} P(t_s + t_h)2 + \frac{N^2}{P} \operatorname{ld} \sqrt{P} 2 t_w + \frac{N^2}{P} \frac{t_e}{2}$$

Recursive Transposition Algorithm

Program (Recursive transposition algorithm on hypercube)

parallel *recursive transpose*
{
  const int $d = \dots, n = \dots;$
  const int $P = 2^d, N = 2^n;$

  process Π[**int** $(p, q) \in \{0, \dots, 2^d - 1\} \times \{0, \dots, 2^d - 1\}$]
  {
    Matrix *A, B;* *// A is the input matrix*
    void *rta(***int** *r,* **int** *s,* **int** *k)*
    {
      if $(k == 0)$ { $A = A^T$; **return**; }
      int $i = p - r, j = q - s, l = 2^{k-1};$
      if $(i < l)$
      {
        if $(j < l)$
        { *// left upper*
          recv$(B,\Pi_{p+l,q})$; **send**$(B,\Pi_{p,q+l})$;
          rta$(r,s,k-1)$;
        }
        else
        { *// right upper*
          send$(A,\Pi_{p+l,q})$; **recv**$(A,\Pi_{p,q-l})$;
          rta$(r,s+l,k-1)$;
        }
      }
      ...
}

Recursive Transposition Algorithm cont.

Program (Recursive transposition algorithm on hypercube cont.)

```
parallel recursive transpose cont.
{
            ...
            else
            {
                if (j < l) {                          // left lower
                    send(A,Π_{p−l,q}); recv(A,Π_{p,q+l});
                    rta(r + l,s,k − 1);
                }
                else
                {                                     // right lower
                    recv(B,Π_{p−l,q}); send(B,Π_{p,q−l});
                    rta(r + l,s + l,k − 1);
                }
            }
        }
        rta(0,0,d);
    }
}
```