Communication Avoiding Algorithms in Dense Linear Algebra

Ting Lin, 1700010644

June 29, 2020

Contents

In this report we introduce communication avoiding algorithms, which is emerging but powerful computational methods designed for fast computation and expansive communication machine. We focus on dense linear algebra, specifically LU and QR factorization. We show the algorithms, model analysis, and experiment performance. Also, some background information, discussions and possible future works are displayed.

1 Introduction

Algorithms often have two parts of costs: Computation and Communication. Communication can happen in move data between either different levels of memory (in sequential case) or processors through network (in parallel case). The (time) cost in communication can be modeled by **alpha-beta** model, which means that we move n words in one message will take $\alpha + \beta n$ time unit, where α is latency and β is the inverse of bandwidth. Moreover, we use γ to denote cost time per FLOP. Typically, we have

$$\gamma << \beta << \alpha,$$

and the gap grows exponentially, inspired by Moore's law. A good algorithm, even in sequential case, should minimize the communication cost.

In past two decades, communication avoiding algorithms, including [1, 2, 3], have been proposed and well studied. Most of them are superior in theory and practice. In this report, we introduce communication avoiding algorithm in dense linear algebra. We will give several reasons why we focus on dense linear algebra:

1. Contrast with Krylov subspace method, dense linear algebra, including LU factorization and QR factorization are easy to analyze its time/energy cost and stability.
2. Unlike CA algorithms in Sparse linear algebra, algorithms in dense algebra always attains its theoretical lower bound, meaning that we achieve an optimal framework in theory.
3. In data science, a faster solver in LU factorization and QR factorization is urgently wanted. Especially for some random methods like random SVD, a lot of techniques for tall and skinny matrices need to be further developed.

Hence we first focus on dense linear algebra to get some feelings in communication avoiding algorithms. This does not mean that CA methods in sparse linear algebra can be ignored since it is crucial in scientific computing. Existing CA improvement in sparse linear algebra including

- Minimize communication in SpMV (Sparse Matrix-Vector Multiplication). See [5, 6].
- Rearrange the computation step in traditional KSM. For example, we postpone the orthogonalization step until the whole Krylov matrix is obtained. Hence a TSQR algorithm can reduce the communication cost. See [7] for further details.
- Change the subspace used in projection methods, for example Enlarged KSM. See [8]

What's more, communication avoiding algorithms arises in several domains, e.g., eigenvalue solver ([9]), preconditioner design ([10]), optimization ([11]) and machine learning ([12, 13]). We will see that in most domain communication avoiding algorithms are far too unexplored and deserve further study.

2 Tall Skinny QR and Communication Avoiding QR Factorization

We first introduce QR factorization, introduced by [1]. We begin with TSQR, which performs with many more rows than columns. Based on TSQR, we introduce CAQR for general matrices (here we assume that processors grid dimension is $P_r \times P_c$). We will show in table that the latency and communication cost is lower than existing algorithms in LAPACK and ScaLAPACK, with th same numerical stability.

2.1 TSQR

The basic idea of parallel TSQR is reduction QR on a binary tree. Suppose we decompose $m \times n$ matrix A into $m/4 \times n$ block rows:

$$A = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix}.$$

We first compute QR decomposition for each block

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} Q_{00}R_{00} \\ Q_{10}R_{10} \\ Q_{20}R_{20} \\ Q_{30}R_{30} \end{pmatrix} \tag{1}$$

We group them into two pairs, R_{00}, R_{10} and R_{20}, R_{30}, and compute the QR decomposition respectively.

$$\begin{pmatrix} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{pmatrix} = \begin{pmatrix} R_{00}^* \\ R_{20}^* \end{pmatrix} = \begin{pmatrix} Q_{01}R_{01} \\ \hline R_{11}R_{11} \end{pmatrix} \tag{2}$$

where

$$R_{00}^* = \begin{pmatrix} R_{00} \\ R_{10} \end{pmatrix}, R_{20}^* = \begin{pmatrix} R_{20} \\ R_{30} \end{pmatrix}.$$

For general size, TSQR can be defined recursively along the binary tree. As shown in Matrix Computation, TSQR is as stable as Householder QR, while other QR methods (e.g., Variants of Gram–Schmidt orthogonalization, CholeskyQR) are not so stable.

2.2 More efficient QR factorization for structured matrices

In the parallel case, all local QR factorization but the zeroth layer have a particular structure: they are (q, typical 2) vertical stacks of $n \times n$ upper triangular matrices (x denotes some non-zero elements):

$$\begin{pmatrix} R_0 \\ R_1 \end{pmatrix} = \begin{pmatrix} x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & & x & x \\ & & & & x \\ x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & & x & x \\ & & & & x \end{pmatrix} \tag{3}$$

Since the upper $n \times n$ sub-block is already upper triangular, we propose the following householder QR for these structured matrices.

We can find that the total cost is about $\frac{2}{3}(q-1)n^3$ flops, about 1/3 of the original unstructured Householder QR.

Traditional Householder QR is dominated by BLAS 2 operations, which leads to low efficiency in general. YT representations proposed by Schreiber and van Loan in [14] uses BLAS 3 operation to store the householder reflection:

Y has no additional computation and storage cost, and T needs about $\frac{2}{3}n^3$ additional flops and $n(n-1)/2$ additional storage per processor. An interesting attribute is that from the above algorithm we can see that we can postpone the computation of T until necessary, only send Y and τ will reduce the communication cost.

Algorithm 1 Structured Householder QR

```
1: for $j = 1$ to $n$ do
2:     Let $\mathcal{I}_j$ be the index set $\{j, n+1 : n+j, \cdots, (q-1)n+1 : (q-1)n+j\}$.
3:     $w := A(\mathcal{I}_j, j)$
4:     $[\tau_j, v] := Householder(w)$ such that $v(1) = 1$
5:     $X := A(\mathcal{I}_j, j+1 : n)$
6:     $X := (I - \tau_j v v^T)X$
7:     $A(\mathcal{I}_j \setminus \{j\}, j) := v(2 : end)$
8:     $A(\mathcal{I}_j, j+1 : n) = X$
9: end for
```

Algorithm 2 YT representation

```
Input: $n$ Householder reflectors $\rho_j = I - \tau_j v_j v_j^T$
1: for $j = 1$ to $n$ do
2:     if $j == 1$ then
3:         $Y := [v_1]$
4:         $T := [-\tau_1]$
5:     else
6:         $z := -\tau_j(T(Y^T v_j))$
7:         $Y := [Y v_j]$
8:         $T := \begin{bmatrix} T & z \\ 0 & -\tau_j \end{bmatrix}$
9:     end if
10: end for
```

2.3 CAQR based on TSQR and BLAS 3 Householder

CAQR is a recursive methods based on TSQR and BLAS 3 Householder transform. Suppose we want to factorize $P_r \times P_c$ block matrix C, with each block's data (assumed is a $b \times b$ matrix) is stored in one processor. Intuitively speaking, we first perform TSQR in leftmost column, and then update the local C matrices for the rest processors. We call this step **trailing matrix update** and describe it before the whole CAQR algorithm.

Our goal is to perform the operation

$$\begin{pmatrix} R_0 & C'_0 \\ R_1 & C'_1 \end{pmatrix} = \begin{pmatrix} QR & C'_0 \\ & C'_1 \end{pmatrix} = Q \cdot \begin{pmatrix} R & \hat{C}'_0 \\ & \hat{C}'_1 \end{pmatrix} \tag{4}$$

in which Q is the local Q factor and R is the local R factor of $[R_0; R_1]$, and C'_0, C'_1 is the local matrices (called trailing matrix hereafter) before updating, and $\hat{C}'_0, \hat{C}'_1$ are those after updating. By YT representation we have

$$\begin{pmatrix} \hat{C}'_0 \\ \hat{C}'_1 \end{pmatrix} := \left(I - \begin{pmatrix} I \\ Y_1 \end{pmatrix} \cdot T^T \cdot \begin{pmatrix} I \\ Y_1 \end{pmatrix}^T \right) \begin{pmatrix} C'_0 \\ C'_1 \end{pmatrix} \tag{5}$$

Here, we suppose that Y_1 is on processor P_1, C'_0 is on processor P_2, and C'_1 is on processor P_3. We propose a straightforward but suboptimal schedule.

P_1's tasks:

- Send Y_1 and τ to P_3

P_2's tasks:

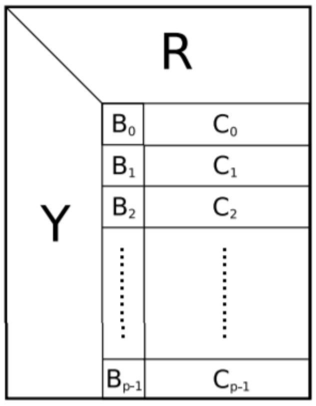
- Send C'_0 to P_3
- Receive W from P_3
- Compute $\hat{C}'_0 = C'_0 - W$

P_3's tasks:

- Receive Y_1 and τ from P_1
- Compute T from Y_1 and τ by Algorithm 2.
- Compute $W = T^T(C'_0 + Y_1^T C'_1)$
- Send W to P_2
- Compute $\hat{C}'_1 = C'_1 - Y_1 W$

The schedule leads to some load imbalance, since P_3 performs much more computation than P_2. But this will be alleviated if we overlap the communication and computation in CAQR algorithm.

CAQR is defined inductively. We assume that first $j-1$ iterations have been performed. Hence Y part and R part is inactive now. We first use TSQR to factorize the stacked block $B_0, \cdots B_{p-1}$, and then update the trailing matrices in $C_0, \cdots, C_{p-1}$. Here we use a butterfly-like techniques to make sure that trailing matrix updating is compatible with TSQR step. We show that in 1 iteration how we update the trailing matrix and obtain the desired factor. Here we suppose we deal with matrix of size $m \times n = P_r b \times P_c b$. For simplicity, we denote the processors in leftmost column are indexed with $0, 1, P_r - 1$. We use several notations to describe our bufferfly-like strategy.



- $level(i,k) = [i/2^k]$ denotes the node at level k of the reduction. The initial stage is $i = 0$, i.e., with no communication.
- $first\_proc(i,k) = 2^k level(i,k)$ is the index of first processor in the node that i belongs to.
- $target(i,k) = first\_proc(i,k) + (i + 2^{k-1}) \mod 2^k$ is the index of the processor with which i exchange data at level k of the butterfly all-reduction algorithm.
- $target\_first\_proc(i,k) = target(first\_proc(i,k))$

We first restate TSQR more clearly, see Algorithm 3

Algorithm 3 TSQR

```
1: For each processor, compute $A_p = Q_{p,0}R_{p,0}$ locally.
2: for $k = 1 : \log P$ do
3:     if $i == target\_first\_proc(i,k)$ or $first\_proc(i,k)$ then
4:         $\phi = first\_proc(i,k), \tau = target\_first\_proc(i,k)$
5:     else
6:         continue
7:     end if
8:     $l = level(i,k)$
9:     $\phi$ and $\tau$ exchange their data.
10:    Compute the structured Householder QR. $\begin{bmatrix} A_\phi \\ A_\tau \end{bmatrix} = Q_{l,k}R_{l,k}$
11: end for
```

Notice that in Algorithm 4 line 5 and line 6, line 8 and line 9 can overlap to reduce the cost time.

2.4 Model Analysis

In this subsection we analyze the cost of TSQR and CAQR, comparing with PDGEQRF in ScaLAPACK.

Algorithm 4 one step CAQR

```
1: Perform TSQR on $[C_0, C_1, \cdots C_{P_r-1}]$, storing $Y_{pk}$ and $\tau_{pk}$ in each processor
2: For each processor in leftmost column, broadcast its $Y_{pk}$ and $\tau_{pk}$ to all processors share the same row with $p$.
3: Form $T_{p,0}$ locally and update the trailing matrix.
4: for $k = 1 : \log P_r$ do
5:   if $p == first\_proc(p,k)$ or $target\_first\_proc(p,k)$ then
6:     Processors in the same row as $target\_first\_proc(p,k)$ form $T_{level(p,k),k}$ locally. Compute $W = Y^T_{level(p,k),k} C_{target\_first\_proc(p,k)}$ locally.
7:     Each processor in the same process row as $first\_proc(p,k)$ sends to the processor in the same column and belonging to the row of $target\_first\_proc(p,k)$, the $C_{first\_proc(p,k)}$.
8:     Compute
           $$W = T^T_{\text{level}(p,k),k} \left(C_{\text{first\_ proc}(p,k)} + W\right) \quad (6)$$
       locally.
9:     Send back $W$ from the row $target\_first\_proc(p,k)$ to $first\_proc(p,k)$ the local $W$.
10:    Update the $target\_first\_proc(p,k)$'s trailing matrix: $C_{target\_first\_proc(p,k)} = C_{target\_first\_proc(p,k)} - Y_{level(p,k),k} W$
11:    Update the $first\_proc(p,k)$'s trailing matrix: $C_{first\_proc(p,k)} = C_{first\_proc(p,k)} - W$
12:  end if
13: end for
```

TSQR A parallel TSQR factorization on a binary reduction tree (just what we introduced) performs the following computations along the critical path:

- One local QR of fully dense $m/P \times n$ matrix $\rightarrow$ ($2mn^2/P - 2n^3/3 + O(mn/P)$ FLOPs)
- $\log P$ structured Householder QR, each $\rightarrow \frac{2}{3}n^3 + O(n^2)$ FLOPs)

Thus the total flop count is

$$\frac{2mn^2}{P} - \frac{2n^3}{3} + \frac{2}{3}n^3 \log P + O\left(\frac{mn}{P}\right). \tag{7}$$

the total message count is $\log P$, the total words communicated is $\log P \frac{n(n+1)}{2}$.

Updating trailing matrix We model the performance of the parallel CAQR algrithms. We assume no overlapping of computation and communication. We recall our assumption: the matrix A is stored in 2-D block cyclic layout on $P_r \times P_c$ grid of processors, and we assume P_r evenly divides m and P_c evenly divides n. Each block is $b \times b$ matrix.

We start counting FLOPs in updating the trailing matrix. First we consider the cost forming T from Y and τ: from $T(1:j-1, 1:j-1)$ to $T(1:j, 1:j)$ we need to compute $\tau T(1:j-1,1:j-1)Y_1(1:j-1,1:j-1)^T v_j(n+1:n+j)$ leading to a cost of $(j-1)(j) + 2\frac{(j-1)^2}{2}$. Summing up j from 1 to n we obtain the FLOPS of forming T is about $\frac{2}{3}n^3 + l.o.t.$ Next we counts the FLOPs in updating trailing matrix:

$$\begin{pmatrix} \hat{C}'_0 \\ \hat{C}'_1 \end{pmatrix} := \left(I - \begin{pmatrix} I \\ Y_1 \end{pmatrix} \cdot T^T \cdot \begin{pmatrix} I \\ Y_1 \end{pmatrix}^T \right) \begin{pmatrix} C'_0 \\ C'_1 \end{pmatrix} \tag{8}$$

in which Y_1 is $n \times n$ upper triangular and C'_0, C'_1 is $n \times q$. We outline the costs which is $3n^2q + 6nq$ in total:

- Compute $W = T^T(C'_0 + Y_1^T C'_1) \rightarrow n(n+1)q + nq + n(n+1)q$ FLOPs

- Compute $\hat{C}'_0 = C'_0 - W \rightarrow nq$ FLOPs
- Compute $\hat{C}'_1 = C'_1 - Y_1 W \rightarrow n(n+2)q$ FLOPs

CAQR We are ready for CAQR. suppose $m_j = (m-1+j)b, n_j = (n-1+j)b$ be the size of active matrix in the $j-th$ iteration

1. TSQR on leftmost column:
   #FLOPs $\approx 2b^3 + \frac{2b^3}{3}\log P_r$ #messages $\approx \log P_r$ #words $\approx \frac{b^2}{2}\log P_r$
2. Broadcasting Householder vector and multipliers along rows:
   #FLOPs ≈ 0 #messages $\approx \log P_c$ #words $\approx (\frac{b^2}{2}+b)\log P_r \log P_c$
3. First update the trailing matrix locally
   #FLOPs $\approx \frac{11}{3}b^3$ #messages ≈ 0 #words ≈ 0
4. Butterfly all-reduce at stage k,$p == first\_proc(p,k)$ or $target\_first\_proc(p,k)$:
   (a) Processors in the same row as $target\_first\_proc(p,k)$ form $T_{level(p,k),k}$ locally. Compute $W = Y^T_{level(p,k),k} C_{target\_first\_proc}$
   locally.
   #FLOPs $\approx \frac{5}{3}b^3$ #messages ≈ 0 #words ≈ 0
   (b) Each processor in the same process row as $first\_proc(p,k)$ sends to the processor in the same column and belonging to the row of $target\_first\_proc(p,k)$, the $C_{first\_proc(p,k)}$.
   #FLOPs 0 #messages 1 #words b^2
   (c) Compute
   $$W = T^T_{\text{level}(p,k),k}\left(C_{\text{first\_ proc}(p,k)} + W\right) \tag{9}$$
   locally.
   #FLOPs $\approx b^3$ #messages 0 #words 0
   (d) Send back W from the row $target\_first\_proc(p,k)$ to $first\_proc(p,k)$ the local W.
   #FLOPs 0 #messages 1 #words b^2
   (e) Update the $target\_first\_proc(p,k)$'s trailing matrix: $C_{target\_first\_proc(p,k)} = C_{target\_first\_proc(p,k)} - Y_{level(p,k),k}W$
   (f) Update the $first\_proc(p,k)$'s trailing matrix: $C_{first\_proc(p,k)} = C_{first\_proc(p,k)} - W$
   #FLOPs $\approx b^3$ #messages 0 #words 0

In total, we have #FLOPs $\approx 2b^3 P_r + 8b^3 P_r \log P_r$ #messages $\approx P_r(3\log P_r + \log P_c)$ #words $\approx \frac{5b^2}{2}P_r \log P_r + \frac{b^2}{2}P_r \log P_r \log P_c$.

2.5 Performance test

We select the experiment results in [1]. Consider a distributed memory, in-DRAM on each node. Two machines are used :

1. **Pentium III cluster (Beowulf)**

   Operated by the University of Colorado at Denver and the Health Sciences Center

   35 dual-socket 900 MHz Pentium III nodes with Dolphin interconnect

   Floating-point rate: 900 Mflop/s per processor, peak

   Network latency: less than 2.7 μs, benchmarked

   Network bandwidth: 350 MB/s, benchmarked upper bound

2. **IBM BlueGene/L ("Frost")**

   Operated by the National Center for Atmospheric Research

   One BlueGene/L rack with 1024 700 MHz compute CPUs

   Floating-point rate: 2.8 Gflop/s per processor, peak

   Network5 latency: 1.5 μs, hardware

   Network one-way bandwidth: 350 MB/s, hardware

Notice that in Table 13 and 14, Pentium II have relatively slow processors with a relative low-latency interconnect, while TSQR was optimized for the opposite case of fast processors and expensive communication. However, it still outperforms than ScaLAPACK.

For CAQR, we test the best choice of CAQR and PDGEQRF (in ScaLAPACK) in IBM Power 5, Peta and Grid.

3 Communication Avoiding LU Factorization

CALU is proposed in a rather similar way ([15, 16]): we first introduce Tall and Skinny LU factorization and then consider the update step by a butterfly all-reduced algorithm. The latter, called GEPP, will be more technical than we encountered in QR.

3.1 TSLU

We first recall Gaussian Elimination with Partial Pivoting called GEPP. Suppose we have two local A with factorization $\Pi_0 A_0 = L_0 U_0$ and $\Pi_1 A_1 = L_1 U_1$. Then the reduction step we need to compute GEPP about following stacked matrix

$$A = \begin{bmatrix} \Pi_0 A_0 \\ \Pi_1 A_1 \end{bmatrix}$$

The algorithmic statement can refer Algorithm **??**.

3.2 Whole CALU and Model Analysis

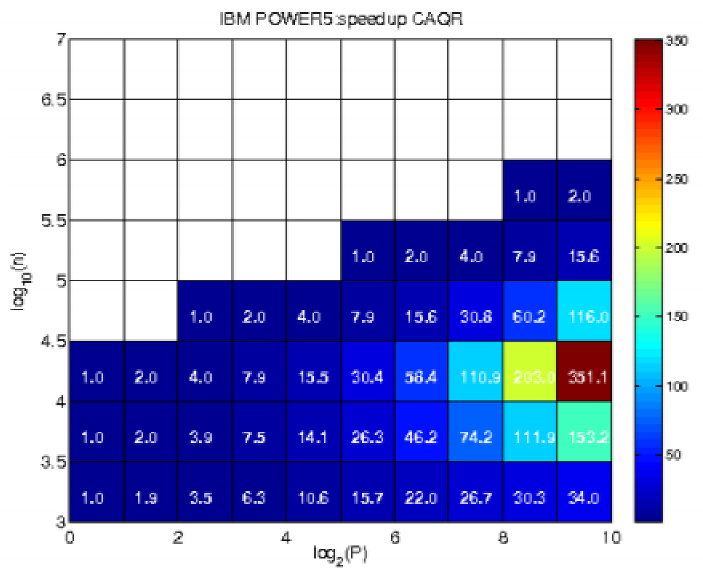
At each iteration, we compute TSLU factorization of the leftmost column. Then we broadcast the pivot information and perform LU iteration.

# procs	CholeskyQR	TSQR (DGEQR3)	CGS	MGS	TSQR (DGEQRF)	ScaLAPACK
1	1.02	4.14	3.73	7.17	9.68	12.63
2	0.99	4.00	6.41	12.56	15.71	19.88
4	0.92	3.35	6.62	12.78	16.07	19.59
8	0.92	2.86	6.87	12.89	11.41	17.85
16	1.00	2.56	7.48	13.02	9.75	17.29
32	1.32	2.82	8.37	13.84	8.15	16.95
64	1.88	5.96	15.46	13.84	9.46	17.74

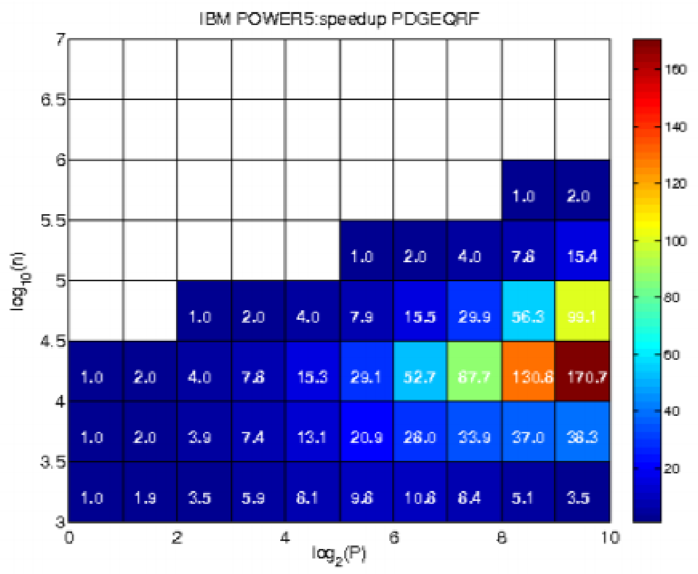
Table 13: Runtime in seconds of various parallel QR factorizations on the Beowulf machine. The total number of rows $m = 100000$ and the ratio $\lceil n/\sqrt{P} \rceil = 50$ (with P being the number of processors) were kept constant

# procs	CholeskyQR	TSQR (DGEQR3)	CGS	MGS	TSQR (DGEQRF)	ScaLAPACK
1	0.45	3.43	3.61	7.13	7.07	7.26
2	0.47	4.02	7.11	14.04	11.59	13.95
4	0.47	4.29	6.09	12.09	13.94	13.74
8	0.50	4.30	7.53	15.06	14.21	14.05
16	0.54	4.33	7.79	15.04	14.66	14.94
32	0.52	4.42	7.85	15.38	14.95	15.01
64	0.65	4.45	7.96	15.46	14.66	15.33

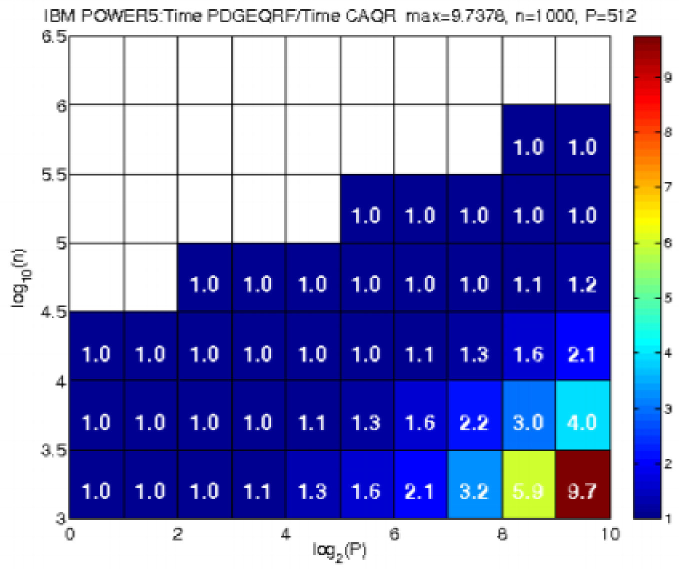
Table 14: Runtime in seconds of various parallel QR factorizations on the Beowulf machine, illustrating weak scaling with respect to the total number of rows m in the matrix. The ratio $\lceil m/P \rceil = 100000$ and the total number of columns $n = 50$ were kept constant as the number of processors P varied from 1



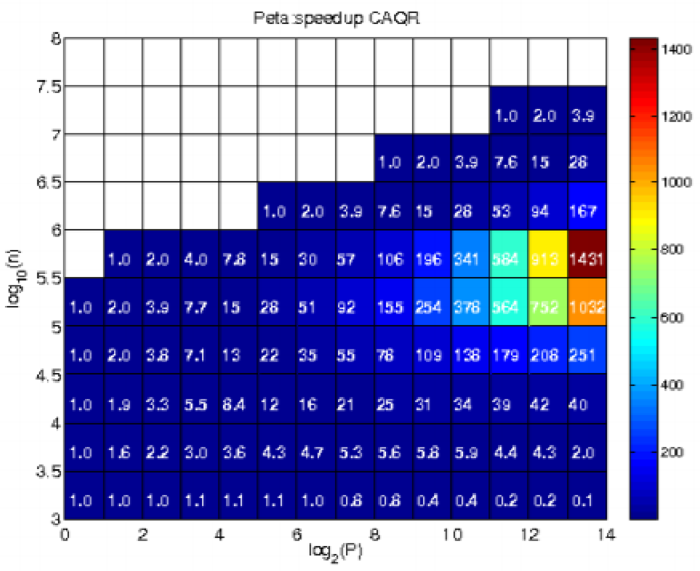
(a) Speedup CAQR



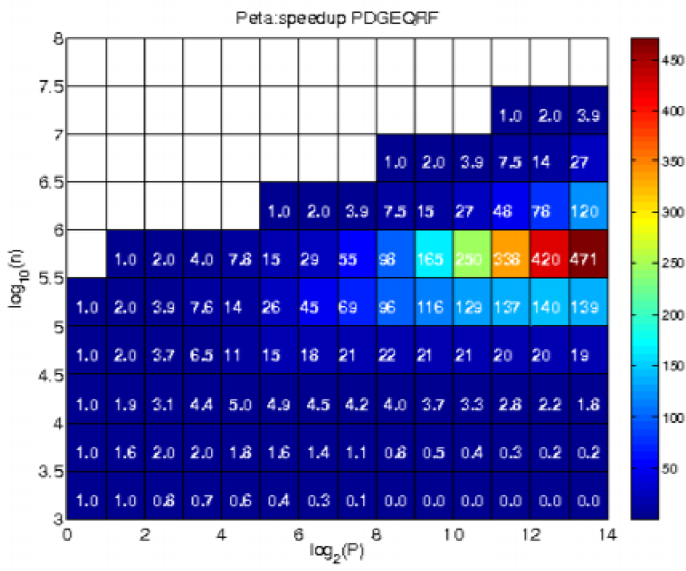
(b) Speedup PDGEQRF



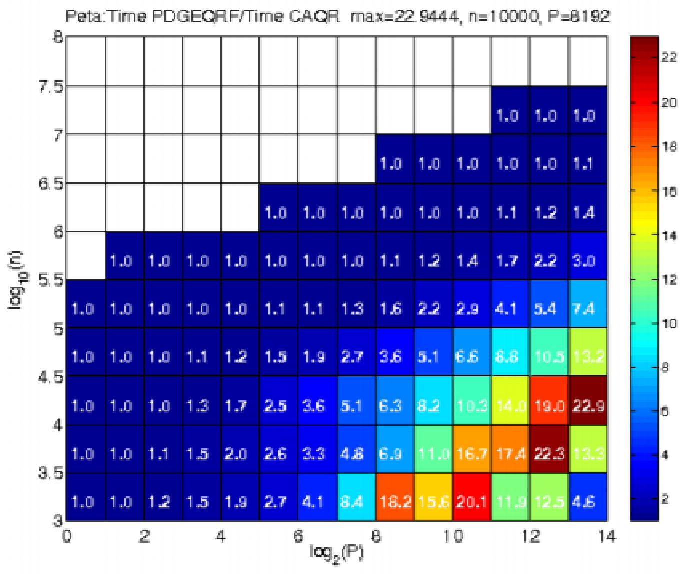
(c) Comparison

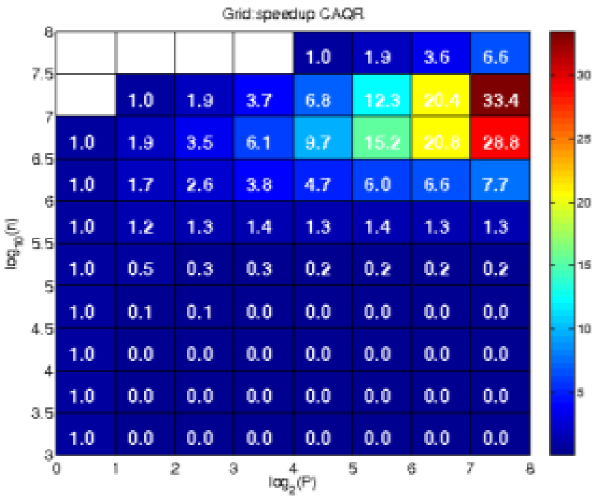


(a) Speedup CAQR

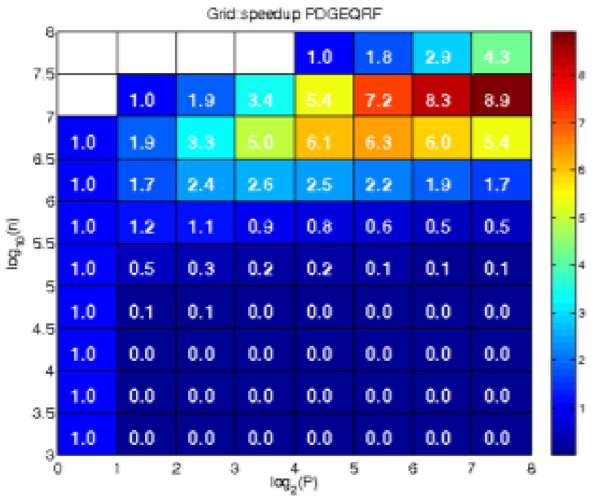


(b) Speedup PDGEQRF

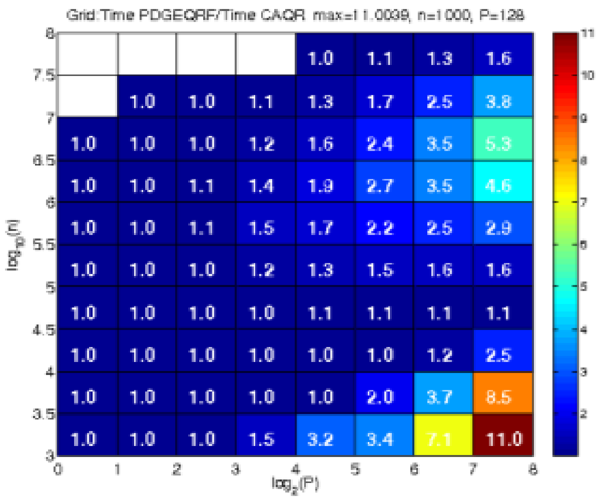




(a) Speedup CAQR



(b) Speedup PDGEQRF



(c) Comparison

Algorithm 5 TSLU

```
1: For each processor, compute $\Pi_{p,0}A_p = L_{p,0}U_{p,0}$ locally. Let $B_i = (\Pi_{p,0}A_i)(1:b, 1:b)$
2: for $k = 1 : \log P$ do
3:   if $i \geq target\_first\_proc(i,k)$ then
4:     $\phi = target(i,k), \tau = i$
5:   else
6:     $\tau = target(i,k), \phi = i$
7:   end if
8:   $l = level(i,k)$
9:   $\phi$ and $\tau$ exchange their data.
10:  Compute the LU decomposition. $\Pi_{l,k}\begin{bmatrix}B_\phi\\B_\tau\end{bmatrix} = L_{l,k}R_{l,k}$
11:  Let $B_i = \left(\Pi_{l,k}\begin{bmatrix}B_\phi\\B_\tau\end{bmatrix}\right)(1:b, 1:b)$
12: end for
13: The final permutation is $\Pi = \Pi_{\log P}\cdots\Pi_0$
14: $U = U_{0,\log P}$
15: solve $L = AU^{-1}$ locally.
```

Algorithm 6 one step CALU

```
1: Perform TSLU on the leftmost block column.
2: Broadcast the pivot information into all processors. For all processors in leftmost column, broadcast its $L$ along the processor row.
3: Swap rows according to pivot information.
4: For all processors in uppermost row, update trailing matrix $A = L^{-1}A$. Then broadcast it along processor column, denoted as $U$.
5: For all rest processors, update $A = A - LU$.
```

3.3 Model Analysis

We summarize the cost of TSLU and CALU for short, since the counting is similar to those in QR, again we omit all lower order terms.

For TSLU, we need about $2mb^2/P + \frac{2b^3}{3}(\log P)$ FLOPs, $\log P$ messages , and $\frac{b^2}{2}\log P$ words to communicate.

For CALU, we need about $2b^3P_r + \frac{2b^3}{3}P_r(\log P_r) + 4b^2P_r\log P_r$ FLOPs,$P_r(2\log P_r + 2\log P_c)$ messages, $b^2P_r(\log P_r + \log P_c)$ words to communicate.

3.4 Performance

We show the result in [16] and references therein, see Table 3 and 4.

4 Discussions and Possible Future Work

Communication Avoiding algorithms express great power in both theory and practice. However, even for dense linear algebra (which we attained lower bound already), we still have a lot of research topic: for example implementation of CA algorithm into various multi-/many- core architecture and improve the performance specifically, see some extension work on GPU [17, 18], FPGA [19, 20]. Some work also use re-scheduling and pipeline to enhance performance intensively, like [4]. What's more, we can utilize our well-developed algorithms in data analysis and linear solver. For example in SVD and low rank approximation. For other future possible work includes numerical scheme and preconditioner design in scientific computing. For example framework of finite volume methods and domain decomposition methods.

References

[1] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. 34(1):A206–A239.

[2] Laura Grigori and Inria Rocquencourt. Introduction to communication avoiding linear algebra algorithms in high performance computing. page 11.

[3] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. 23:1–155.

[4] E. Georganas, J. Gonzalez-Dominguez, E. Solomonik, Y. Zheng, J. Tourino, and K. Yelick. Communication avoiding and overlapping for numerical linear algebra. In SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–11, 2012.

[5] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In 2008 IEEE International Symposium on Parallel and Distributed Processing, pages 1–12. IEEE. ISSN: 1530-2075.

[6] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. Minimizing communication in sparse matrix solvers. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pages 1–12. ISSN: 2167-4337.

[7] Mark Frederick Hoemmen. Communication-avoiding krylov subspace methods. page 358.

IBM POWER 5											
		No of processors $P = P_r \times P_c$									
m	$n = b$	4		8		16		32		64	
		2×2		2×4		4×4		4×8		8×8	
		Rec	Cl	Rec	Cl	Rec	Cl	Rec	Cl	Rec	Cl
10^3	50	**1.66**	1.59	1.96	**2.06**	**2.24**	2.09	-	-	-	-
10^3	100	1.27	1.17	1.44	1.37	-	-	-	-	-	-
10^3	150	1.06	0.97	-	-	-	-	-	-	-	-
$5 \cdot 10^3$	50	**1.62**	1.08	**1.48**	1.44	**1.97**	1.94	1.78	**2.05**	**2.09**	1.71
$5 \cdot 10^3$	100	0.98	0.85	1.13	1.04	1.36	1.29	1.39	1.47	-	-
$5 \cdot 10^3$	150	1.08	0.81	1.00	1.01	1.06	0.96	0.99	0.97	-	-
10^4	50	1.24	0.87	**1.71**	0.88	**1.78**	1.68	1.66	**1.94**	**2.18**	1.76
10^4	100	1.34	0.81	1.01	0.80	1.26	1.15	1.30	1.28	1.51	1.21
10^4	150	**3.07**	0.88	1.03	0.78	1.01	0.89	1.00	0.97	1.06	0.80
10^5	50	1.07	0.70	1.09	0.72	1.18	0.85	1.15	**1.32**	**1.50**	1.23
10^5	100	1.00	0.70	1.04	0.67	1.09	0.73	1.21	1.03	1.19	1.01
10^5	150	**1.13**	0.68	**1.13**	0.69	**1.36**	0.77	1.08	0.84	1.03	0.75
10^6	50	1.36	0.71	1.27	0.71	1.25	0.70	1.12	0.69	**2.01**	0.82
10^6	100	1.84	0.75	1.95	0.87	1.62	0.73	2.90	0.84	1.08	0.70
10^6	150	**2.32**	0.81	**2.34**	0.89	**4.37**	0.90	**3.42**	0.85	1.22	0.70

Cray XT4											
		No of processors $P = P_r \times P_c$									
m	$n = b$	4		8		16		32		64	
		2×2		2×4		4×4		4×8		8×8	
		Rec	Cl	Rec	Cl	Rec	Cl	Rec	Cl	Rec	Cl
10^3	50	1.42	**2.23**	1.85	**2.71**	2.09	**3.09**	-	-	-	-
10^3	100	1.14	1.39	1.29	1.56	-	-	-	-	-	-
10^3	150	1.12	0.91	-	-	-	-	-	-	-	-
$5 \cdot 10^3$	50	1.22	1.42	1.65	**2.15**	1.97	**2.72**	2.10	**3.06**	1.04	**2.59**
$5 \cdot 10^3$	100	1.27	1.24	1.25	1.32	1.35	1.53	1.38	1.65	-	-
$5 \cdot 10^3$	150	**1.67**	1.22	0.97	0.90	0.88	0.91	0.85	0.90	-	-
10^4	50	1.20	1.14	1.37	1.19	1.85	**2.42**	1.03	**2.88**	2.03	**3.10**
10^4	100	2.19	1.34	1.56	1.44	1.30	1.41	0.94	1.56	1.36	1.94
10^4	150	**2.61**	1.30	**2.03**	1.44	0.92	0.88	0.81	0.90	0.87	0.93
10^5	50	2.12	1.13	2.23	1.37	2.29	1.50	1.20	1.47	1.76	1.88
10^5	100	3.14	1.25	3.13	1.45	2.97	1.43	1.92	1.39	**2.38**	1.39
10^5	150	**3.78**	1.30	**3.57**	1.47	**3.14**	1.30	**2.12**	1.26	2.34	1.15
10^6	50	2.99	1.49	3.02	1.51	2.86	1.28	2.09	1.03	2.14	1.31
10^6	100	4.51	1.65	4.55	1.71	4.04	1.36	3.04	1.13	3.07	1.27
10^6	150	**5.58**	1.61	**5.52**	1.76	**4.80**	1.39	**3.60**	1.12	**3.67**	1.20

Table 3: Time ratio of PDGETF2 to TSLU obtained on IBM POWER 5 and Cray XT4 systems, using DGETF2 for the local LU factorization (Cl), and using RGETF2 for the local LU factorization (Rec). The matrix factorized is $m \times n$, with a block of size $b = n$.

IBM POWER 5											
		No of processors $P = P_r \times P_c$									
$m = n$	b	4		8		16		32		64	
		2×2		2×4		4×4		4×8		8×8	
		Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU
10^3	50	**1.57**	9.79	**1.59**	11.9	**2.23**	11.8	**2.07**	11.5	2.25	9.8
10^3	100	1.48	8.84	1.47	10.5	1.91	10.6	1.91	11.2	**2.29**	10.9
10^3	150	1.36	8.26	1.41	9.9	1.70	9.5	-	-	-	-
$5 \cdot 10^3$	50	1.05	21.5	1.09	39.4	**1.31**	61.2	1.51	95.5	**1.69**	118.6
$5 \cdot 10^3$	100	**1.06**	21.1	**1.13**	37.3	1.21	56.7	**1.67**	84.1	1.66	103.1
$5 \cdot 10^3$	150	1.04	20.6	1.08	35.1	1.18	52.3	1.26	74.8	1.45	89.1
10^4	50	1.00	23.27	1.00	45.1	1.08	80.3	1.17	143.2	**1.35**	213.9
10^4	100	1.00	23.62	1.00	44.4	1.10	78.0	1.19	133.2	1.24	197.6
10^4	150	**1.01**	23.47	**1.02**	42.3	**1.33**	74.1	**1.59**	122.1	1.17	173.8

CRAY XT4											
		No of processors $P = P_r \times P_c$									
$m = n$	b	4		8		16		32		64	
		2×2		2×4		4×4		4×8		8×8	
		Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU
10^3	50	1.19	5.4	1.20	6.6	1.33	6.6	1.35	7.5	1.67	7.6
10^3	100	**1.28**	5.5	**1.39**	7.0	**1.52**	7.2	**1.60**	8.5	**1.81**	8.3
10^3	150	1.23	5.3	1.32	6.6	1.44	6.7	-	-	-	-
$5 \cdot 10^3$	50	1.03	19.2	1.09	33.3	1.12	44.3	1.16	69.2	1.11	67.2
$5 \cdot 10^3$	100	1.12	19.4	1.20	32.3	1.13	42.8	1.24	67.4	1.32	76.1
$5 \cdot 10^3$	150	**1.23**	19.1	**1.38**	31.0	**1.22**	40.8	**1.35**	61.9	**1.36**	70.5
10^4	50	1.01	24.4	1.05	45.7	1.04	69.5	1.08	121.3	1.31	154.9
10^4	100	1.09	25.3	1.18	46.5	1.13	69.8	1.22	118.2	**1.33**	153.3
10^4	150	**1.16**	25.2	**1.31**	45.4	**1.22**	67.3	**1.38**	111.4	1.30	140.3

Table 4: Time ratio of PDGETRF to CALU (Impvt columns) and performance for CALU in GFLOPs/s (GFlops columns) obtained on IBM POWER 5 system. The matrix factorized is $m \times m$, with a block of size b.

[8] Laura Grigori, Sophie Moufawad, and Frederic Nataf. Enlarged krylov subspace conjugate gradient methods for reducing communication. 37(2):744–773.

[9] Grey Ballard, James Demmel, and Ioana Dumitriu. Minimizing communication for eigenproblems and the singular value decomposition. Technical Report UCB/EECS-2011-14, EECS Department, University of California, Berkeley, Feb 2011.

[10] Laura Grigori and Sophie Moufawad. Communication avoiding ILU0 preconditioner. 37(2):C217–C246.

[11] Saeed Soori, Aditya Devarakonda, James Demmel, Mert Gurbuzbalaban, and Maryam Mehri Dehnavi. Avoiding communication in proximal methods for convex optimization problems.

[12] Yang You, James Demmel, Kenneth Czechowski, Le Song, and Richard Vuduc. CA-SVM: Communication-avoiding support vector machines on distributed systems. In 2015 IEEE International Parallel and Distributed Processing Symposium, pages 847–859. ISSN: 1530-2075.

[13] Penporn Koanantakool. Communication Avoidance for Algorithms with Sparse All-to-all Interactions. PhD thesis, EECS Department, University of California, Berkeley, Dec 2017.

[14] Robert Schreiber and Charles VanLoan. A storage-efficient wy representation for products of householder transformations. SIAM Journal on Scientific and Statistical Computing, 10, 02 1989.

[15] Laura Grigori, James W. Demmel, and Hua Xiang. Communication avoiding gaussian elimination. In SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pages 1–12. ISSN: 2167-4337.

[16] Laura Grigori, James W. Demmel, and Hua Xiang. CALU: A communication optimal LU factorization algorithm. 32(4):1317–1350.

[17] Michael Anderson, Grey Ballard, James Demmel, and Kurt Keutzer. Communication-avoiding QR decomposition for GPUs. In 2011 IEEE International Parallel Distributed Processing Symposium, pages 48–58. ISSN: 1530-2075.

[18] Maryam MehriDehnavi, Yousef El-Kurdi, James Demmel, and Dennis Giannacopoulos. Communication-avoiding krylov techniques on graphic processing units. 49(5):1749–1752. Conference Name: IEEE Transactions on Magnetics.

[19] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. Flexible communication avoiding matrix multiplication on fpga with high-level synthesis. In The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 20, page 244254, New York, NY, USA, 2020. Association for Computing Machinery.

[20] A. Rafique, N. Kapre, and G. A. Constantinides. Enhancing performance of tall-skinny qr factorization using fpgas. In 22nd International Conference on Field Programmable Logic and Applications (FPL), pages 443–450, 2012.