Efficient Parallel Implementation of Classical Gram-Schmidt Orthogonalization Using Matrix Multiplication

Takuya Yokozawa　Daisuke Takahashi

Taisuke Boku　Mitsuhisa Sato

Graduate School of Systems and Information Engineering,
University of Tsukuba, Japan

Overview

- Background and objective
- Classical Gram-Schmidt orthogonalization
  - Using matrix multiplication
- Proposed algorithm
  - Recursive CGS
- Performance results
- Conclusion and future works

Background

- Gram-Schmidt orthogonalization process is one of the fundamental algorithms in linear algebra
- Two basic computational variants of the Gram-Schmidt process exist:
  - Classical Gram-Schmidt algorithm (CGS)
  - Modified Gram-Schmidt algorithm (MGS)
    - Often selected for practical application
    - More stable than the CGS algorithm

Background (cont'd)

- MGS is stable but,
  - Cannot be expressed by Level-2 BLAS
  - Parallel implementation requires additional communication
- CGS is unstable but,
  - Can be expressed by Level-2 BLAS
  - CGS with DGKS correction [Daniel et al. '76] is one of the most efficient way to perform the orthogonalization process

In this work, we use the CGS

In this work

- We study an efficient implementation of the CGS orthogonalization using matrix multiplication
- We propose a parallel recursive CGS algorithm to perform QR decomposition
- We report some performance results on PC-cluster

Related works

- Recursion leads to automatic variable blocking for dense linear-algebra algorithms [Gustavson 1997]

- Parallel QR factorization
  [Elmroth and Gustavson 2000]
  - Recursive QR factorization of m by n matrix
  - On 4-way SMP computer

Classical Gram-Schmidt algorithm for matrix

- The CGS orthogonalization can be performed by using Level-2 BLAS
- Suitable for parallelization
- The CGS algorithm is not stable

$$
\begin{array}{l}
\text{do } j = 1,n \\
\quad \boldsymbol{q}_j = \boldsymbol{a}_j \\
\quad \text{do } i = 1,j\text{-}1 \\
\quad\quad \boldsymbol{q}_j = \boldsymbol{a}_j - (\boldsymbol{q}_i, \boldsymbol{a}_j)\boldsymbol{q}_i \\
\quad \text{end do} \\
\quad \boldsymbol{q}_j = \boldsymbol{q}_j / \|\boldsymbol{q}_j\| \\
\text{end do}
\end{array}
$$

$\boldsymbol{a}_j$ Raw data vector

$\boldsymbol{q}_j$ Orthogonalized vector

$\|\boldsymbol{q}_j\|$ Euclid norm

$(\boldsymbol{q}_i, \boldsymbol{a}_j)$ Inner product

Naïve implementation

- CGS for m by n matrix is computed by
  - $2m$ Matrix-vector multiplications (GEMV; Level-2 BLAS)
  - m normalizations (NRM2, SCAL; Level-1 BLAS)

$\boldsymbol{A} = (\boldsymbol{a_1} \quad \boldsymbol{a_2} \quad \Lambda \quad \boldsymbol{a_n})$ $\quad \boldsymbol{a_i}$ is vector

$$\boldsymbol{q}_1 = \boldsymbol{a}_1, \qquad \boldsymbol{q}_1 = \boldsymbol{q}_1 / \|\boldsymbol{q}_1\|$$

$$\boldsymbol{q}_2 = \boldsymbol{a}_2 - (\boldsymbol{q}_1, \boldsymbol{a}_2)\boldsymbol{q}_1, \qquad \boldsymbol{q}_2 = \boldsymbol{q}_2 / \|\boldsymbol{q}_2\|$$

$$\boldsymbol{q}_3 = \boldsymbol{a}_3 - (\boldsymbol{q}_1, \boldsymbol{a}_3)\boldsymbol{q}_1 - (\boldsymbol{q}_2, \boldsymbol{a}_3)\boldsymbol{q}_2, \qquad \boldsymbol{q}_3 = \boldsymbol{q}_3 / \|\boldsymbol{q}_3\|$$

$$\boldsymbol{q}_4 = \boldsymbol{a}_4 - (\boldsymbol{q}_1, \boldsymbol{a}_4)\boldsymbol{q}_1 - (\boldsymbol{q}_1, \boldsymbol{a}_4)\boldsymbol{q}_2 - (\boldsymbol{q}_1, \boldsymbol{a}_4)\boldsymbol{q}_3, \qquad \boldsymbol{q}_4 = \boldsymbol{q}_4 / \|\boldsymbol{q}_4\|$$

$$\boldsymbol{q}_5 = \boldsymbol{a}_5 - (\boldsymbol{q}_1, \boldsymbol{a}_5)\boldsymbol{q}_1 - (\boldsymbol{q}_1, \boldsymbol{a}_5)\boldsymbol{q}_2 - (\boldsymbol{q}_1, \boldsymbol{a}_5)\boldsymbol{q}_3 - (\boldsymbol{q}_1, \boldsymbol{a}_5)\boldsymbol{q}_4, \qquad \boldsymbol{q}_5 = \boldsymbol{q}_5 / \|\boldsymbol{q}_5\|$$

$$\boldsymbol{q}_6 = \boldsymbol{a}_6 - (\boldsymbol{q}_1, \boldsymbol{a}_6)\boldsymbol{q}_1 - (\boldsymbol{q}_1, \boldsymbol{a}_6)\boldsymbol{q}_2 - (\boldsymbol{q}_1, \boldsymbol{a}_6)\boldsymbol{q}_3 - (\boldsymbol{q}_1, \boldsymbol{a}_6)\boldsymbol{q}_4 - (\boldsymbol{q}_1, \boldsymbol{a}_6)\boldsymbol{q}_5, \qquad \boldsymbol{q}_6 = \boldsymbol{q}_6 / \|\boldsymbol{q}_6\|$$

GEMV

NRM2, SCAL

CGS using matrix multiplications

- The CGS orthogonalization of a matrix can be changed into a matrix multiplication. [Samukawa '95]
- The orthogonalization processes of the q_4 q_5 q_6 can be separated
  - Depend on q_1 q_2 q_3 and a_4 a_5 a_6
  - Depend on q_4 q_5
  - Normalization

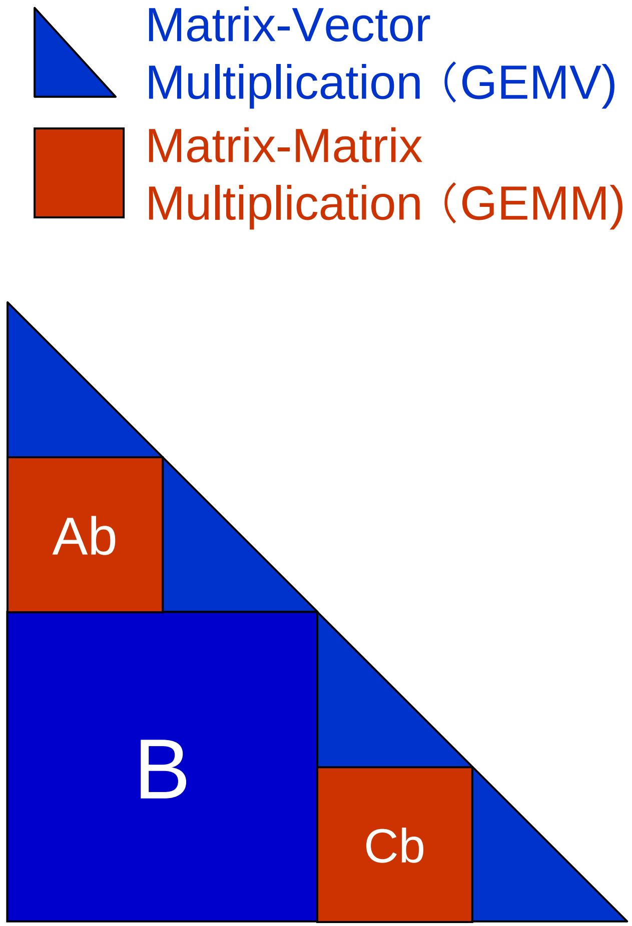
$$q_4 = a_4 - (q_1 \cdot a_4)q_1 - (q_2 \cdot a_4)q_2 - (q_3 \cdot a_4)q_3 \qquad q_4 = q_4 / \|q_4\|$$

$$q_5 = a_5 - (q_1 \cdot a_5)q_1 - (q_2 \cdot a_5)q_2 - (q_3 \cdot a_5)q_3 - (q_4 \cdot a_5)q_4 \qquad q_5 = q_5 / \|q_5\|$$

$$q_6 = a_6 - (q_1 \cdot a_6)q_1 - (q_2 \cdot a_6)q_2 - (q_3 \cdot a_6)q_3 - (q_4 \cdot a_6)q_4 - (q_5 \cdot a_6)q_5 \qquad q_6 = q_6 / \|q_6\|$$

Concept of proposed algorithm

- The CGS orthogonalization can also be extended with matrix multiplication into a recursive formulation
  - The parts of A and C are same structure as original CGS calculation space
  - The parts of Ab and Cb are computed by matrix multiplication



Parallelization

- We use row-wise distribution
  - The proposed recursive algorithm use matrix multiplication and matrix vector multiplication to compute inner products
- Column-wise distribution
  - Whole elements of a vector are in one processor
  - Additional communication is required to compute inner products.
- Row-wise distribution
  - Part of elements of a vector are in each processor
  - Corrective communication is required to sums up and distributes inner products

Recursive CGS

begin recursiveCGS(A,Q,k,m)

  if $(m \le NB)$ then

    $\boldsymbol{q}_k = \boldsymbol{q}_k / \|\boldsymbol{q}_k\|$

    do $j = k+1, k+m$

      $\mathrm{GEMV}(Q^T_{k,j-k}, \boldsymbol{a}_j, \boldsymbol{w})$

      $\mathrm{GEMV}(Q_{k,j-k}, \boldsymbol{w}, \boldsymbol{q}_j)$

      $\boldsymbol{q}_j = \boldsymbol{q}_j / \|\boldsymbol{q}_j\|$

    end do

  else

    recursiveCGS$(A,Q,k,m/2)$

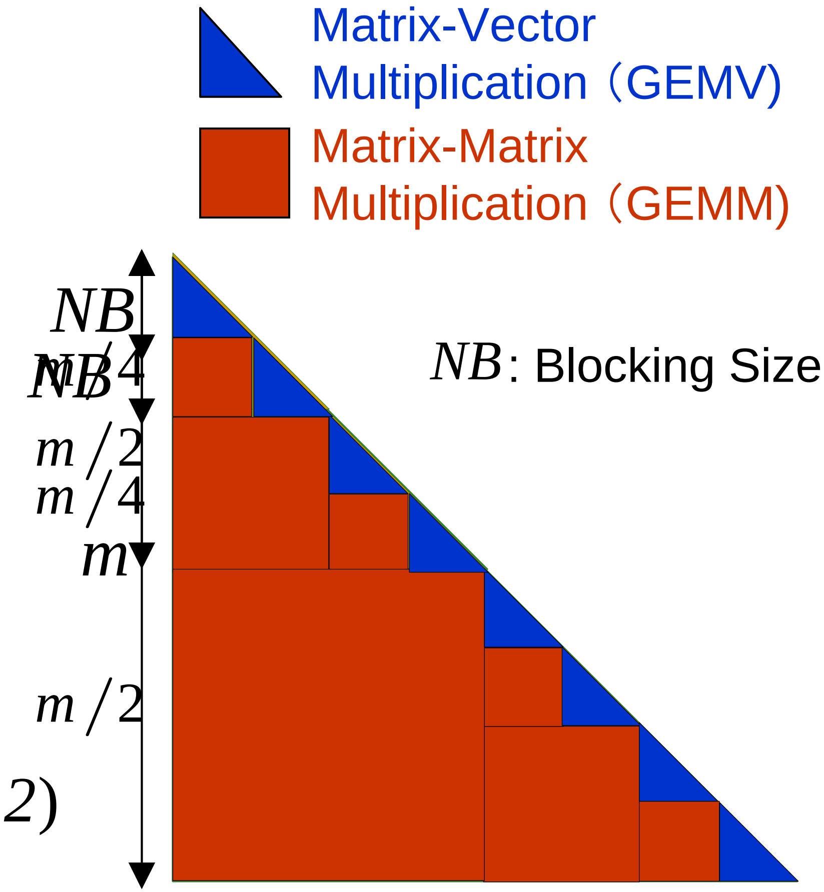
    $\mathrm{GEMM}(Q^T_{k+(m/2),(m/2)}, A_{k+(m/2),(m/2)}, S)$

    $\mathrm{GEMM}(S, Q_{k+(m/2),(m/2)}, Q_{k+(m/2),(m/2)})$

    recursiveCGS$(A,Q,k+m/2,m/2)$

  end if

end



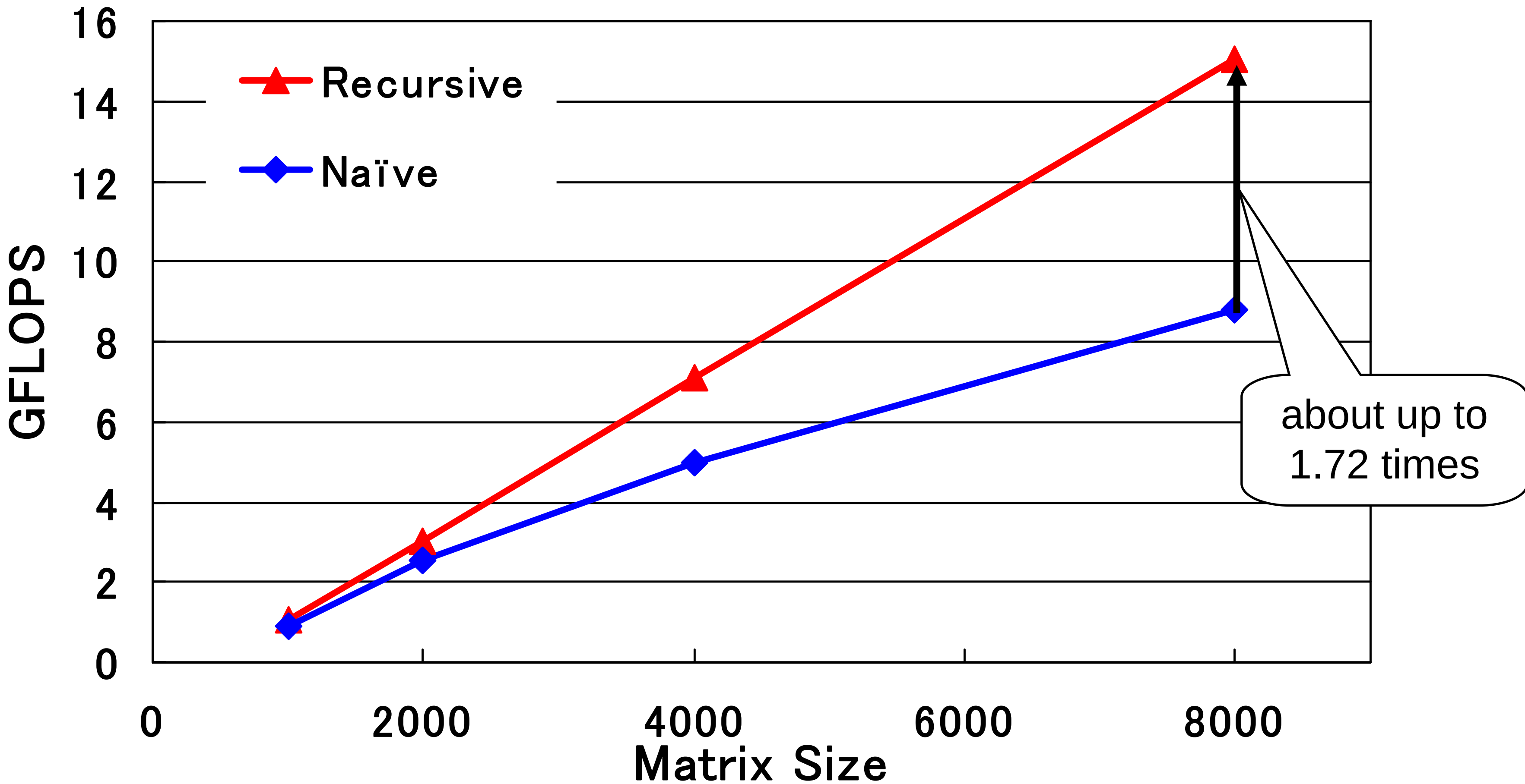
Performance Results

- To evaluate the proposed recursive CGS algorithm, we compared
  - Proposed recursive CGS algorithm
  - Naïve implementation of the CGS algorithm using Level-2 BLAS
- The CGS orthogonalization processes were performed on double-precision real data

Evaluation Environment

- A 32-node Xeon PC-cluster
  - Xeon 3GHz, 1GB DDR2 Memory
- 1000Base-T Gigabit Ethernet
- LAM/MPI 7.1.1
- Intel MKL 8.1
- gcc 4.0.2
- Linux 2.6.17
- All programs were run in 64-bit mode

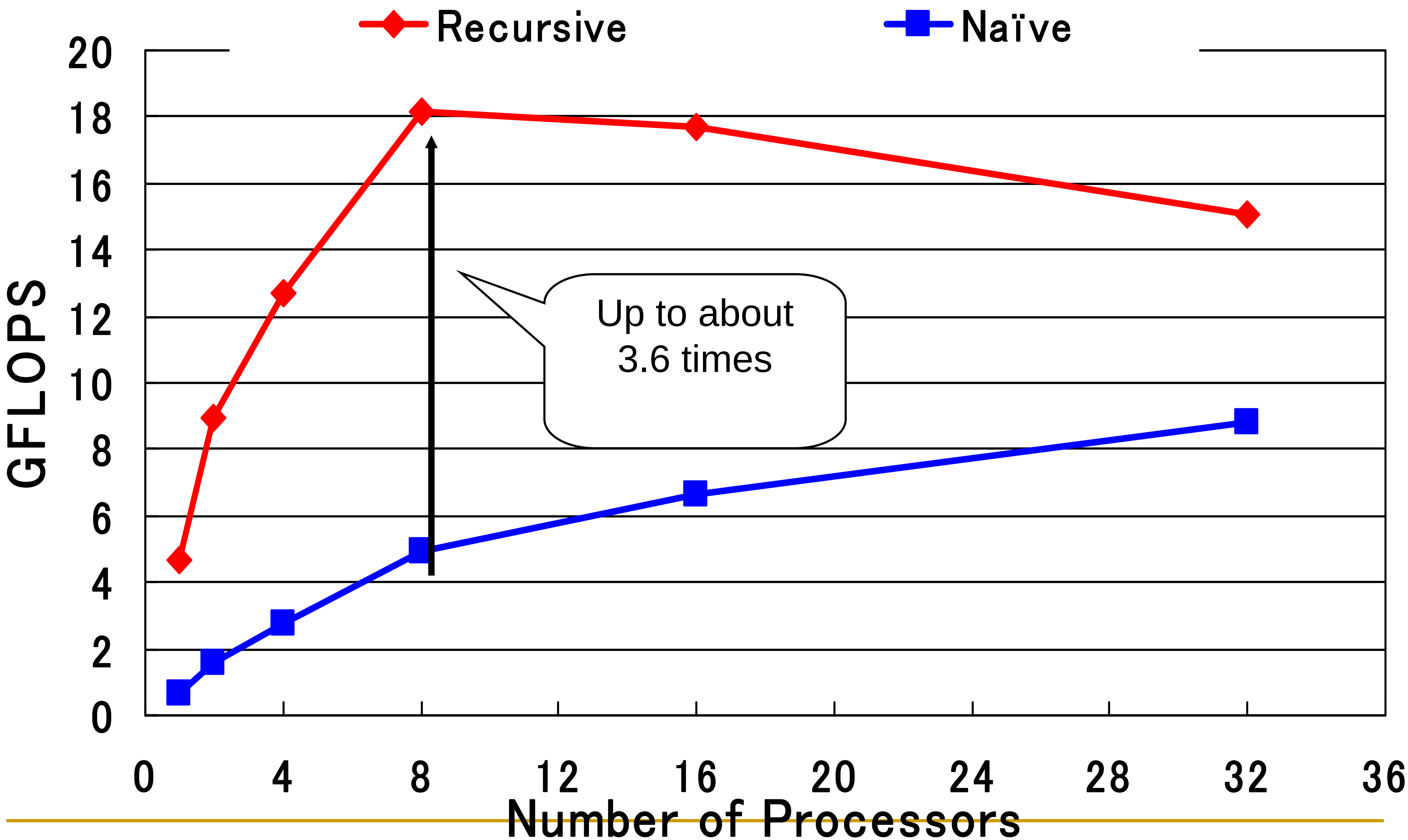
Performance Results on 32-node Xeon PC cluster



Discussion(1)

- The recursive CGS algorithm improved the performance by up to about 1.72 times when matrix size is 8000
- Cache reusability is improved
  - Naive implementation uses Level-2 BLAS
  - The recursive CGS uses Level-3 BLAS

Performance Results (Matrix size N=8000)

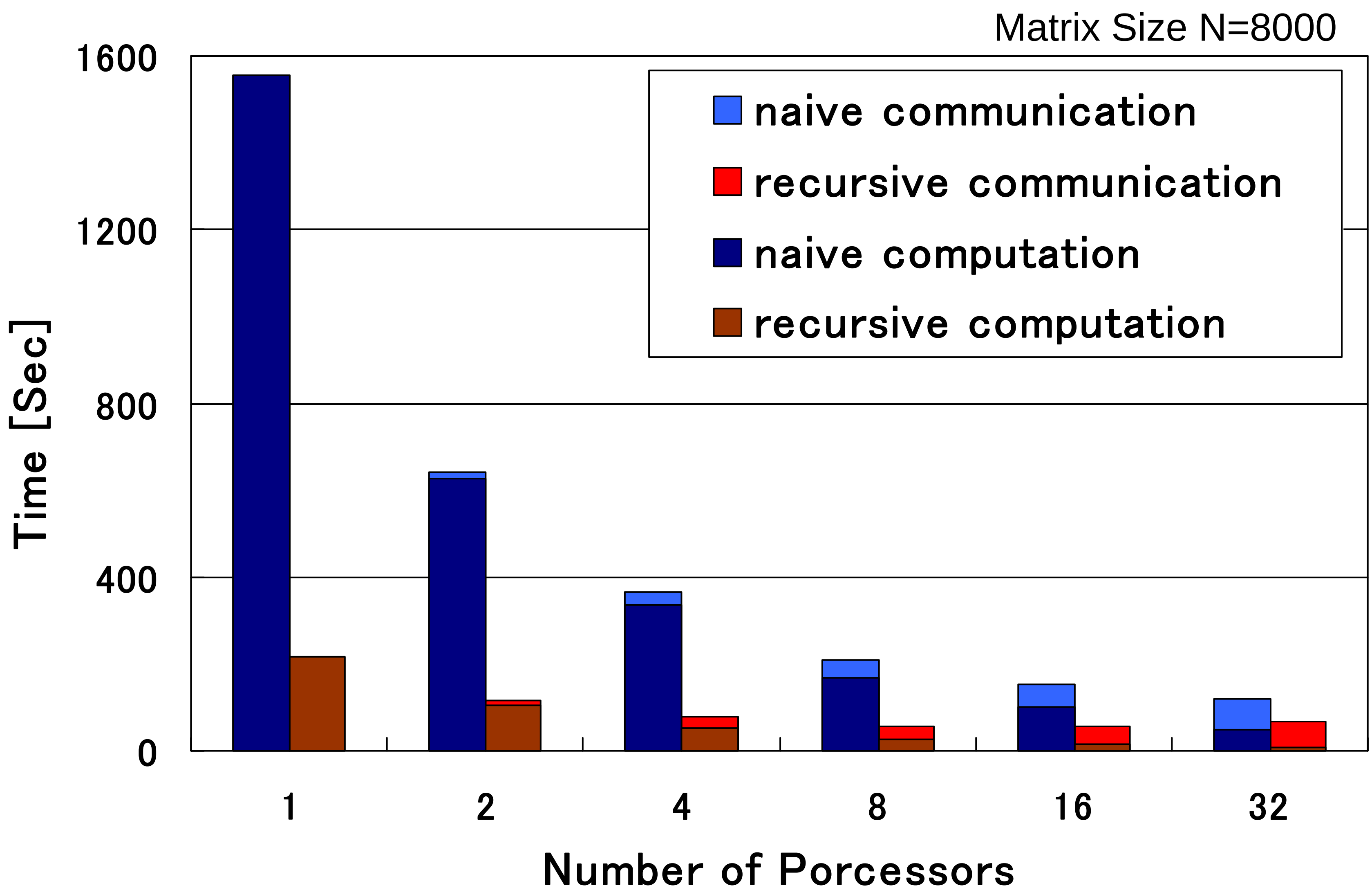


Discussion(2)

- We could improve the performance by up to about 3.6 times when matrix size is 8000

- Scalability is limited
  - MPI_ALLREDUCE communication time dominates in total execution time
  - For larger problem size, matrix multiplications should be distributed

Breakdown of performance results



Conclusion

- We proposed a recursive CGS algorithm using matrix multiplication
- The proposed algorithm improved the performance by naïve implementation
  - Cache reusability was improved since using matrix multiplication
- The performance results demonstrate that the recursive CGS algorithm is efficient for reduce execution time of QR decomposition.

Future works

- Improve scalability
  - Using other data distribution algorithm
- Development algorithm to compute optimal blocking size “NB”