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

\[
\text{do } j = 1, n \\
q_j = a_j \\
\text{do } i = 1, j-1 \\
q_j = a_j - (q_i, a_j)q_i \\
\text{end do} \\
q_j = q_j / \|q_j\| \\
\text{end do}
\]

- \( a_j \) Raw data vector
- \( q_j \) Orthogonalized vector
- \( \|q_j\| \) Euclid norm
- \( (q_i, a_j) \) 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)

$$A = (a_1, a_2, \ldots, a_n) \quad a_i \text{ is vector}$$

- $q_1 = a_1$, $q_1 = q_1/\|q_1\|$  \hspace{1cm} **GEMV**
- $q_2 = a_2 - (q_1, a_2)q_1$, $q_2 = q_2/\|q_2\|$  \hspace{1cm} **NRM2, SCAL**
- $q_3 = a_3 - (q_1, a_3)q_1 - (q_2, a_3)q_2$, $q_3 = q_3/\|q_3\|$  \hspace{1cm} **GEMV**
- $q_4 = a_4 - (q_1, a_4)q_1 - (q_2, a_4)q_2 - (q_1, a_4)q_3$, $q_4 = q_4/\|q_4\|$  \hspace{1cm} **NRM2, SCAL**
- $q_5 = a_5 - (q_1, a_5)q_1 - (q_2, a_5)q_2 - (q_1, a_5)q_3 - (q_1, a_5)q_4$, $q_5 = q_5/\|q_5\|$  \hspace{1cm} **GEMV**
- $q_6 = a_6 - (q_1, a_6)q_1 - (q_2, a_6)q_2 - (q_1, a_6)q_3 - (q_1, a_6)q_4 - (q_1, a_6)q_5$, $q_6 = q_6/\|q_6\|$  \hspace{1cm} **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

\[
q_4 = q_4 / \|q_4\| \\
q_5 = q_5 / \|q_5\| \\
q_6 = q_6 / \|q_6\|
\]
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 ($m \leq NB$) then
    $q_k = q_k / \|q_k\|$
    do $j = k + 1, k + m$
      GEMV($Q^T_{k-j}$,$a_j$,$w$)
      GEMV($Q_{k-j},w,q_j$)
      $q_j = q_j / \|q_j\|$
    end do
  else
    recursiveCGS($A, Q, k, m/2$)
    GEMM($Q^T\|k+(m/2)\|, (m/2), A_{k+(m/2)}, (m/2), S$)
    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

NB: Blocking Size

Matrix-Vector Multiplication (GEMV)
Matrix-Matrix Multiplication (GEMM)
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

![Graph showing performance results for Recursive and Naïve methods]

- Recursive method performs about up to 1.72 times better than the Naïve method.

Matrix Size vs. GFLOPS chart.
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)

- Recursive: Up to about 3.6 times

Graph showing the performance of Recursive and Naïve methods with increasing number of processors. The peaks indicate the performance at different processor counts.
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

Matrix Size N=8000

- naive communication
- recursive communication
- naive computation
- recursive computation

Time [Sec] vs. Number of Processors

- Sep. 7, 06
- PMAA06
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”