FFTE: A High-Performance FFT Library and Parallel Implementation of Classical Gram-Schmidt Orthogonalization

Daisuke Takahashi

Center for Computational Sciences/
Graduate School of Systems and Information Engineering
University of Tsukuba
Outline

• FFTE: A High-Performance FFT Library
  – Features, Design and Approach
  – Performance Results

• Parallel Implementation of Classical Gram-Schmidt Orthogonalization Using Matrix Multiplication (with T. Yokozawa, T. Boku and M. Sato)
  – Column-wise Blocking CGS Algorithm
  – Recursive Blocking CGS Algorithm
  – Performance Results
FFTE: A High-Performance FFT Library

- FFTE is a Fortran subroutine library for computing the Fast Fourier Transform (FFT) in one or more dimensions.
- It includes complex, mixed-radix and parallel transforms.
- FFTE is typically faster than other publically-available FFT implementations, and is even competitive with vendor-tuned libraries.
FFTE Library - Features

• High speed
  – Supports Intel’s SSE2/SSE3 instructions.

• Parallel transforms
  – Shared / Distributed memory parallel computers (OpenMP, MPI and OpenMP + MPI)

• High portability
  – Fortran77 + OpenMP + MPI
  – Intel’s intrinsics for SSE2/SSE3 instructions.

• HPC Challenge Benchmark
  – FFTE’s 1-D parallel FFT routine has been incorporated into the HPC Challenge (HPCC) benchmark.
FFTE Library - Design

• Performance
  – One goal for large FFTs is to minimize the number of cache misses.

• Ease of use: routine interfaces
  – Similar to sequential SGI SCSL or Intel MKL routines

• Portability
  – Communication: MPI
  – Computation: Fortran77 + OpenMP
FFTE Library - Approach

- Many FFT routines work well when data sets fit into a cache.
- When a problem size exceeds the cache size, however, the performance of these FFT routines decreases dramatically.
- Some previously presented three-dimensional FFT algorithms require
  - Three multicolumn FFTs.
  - Three data transpositions.
    → The chief bottlenecks in cache-based processors.
- We combine the multicolumn FFTs and transpositions to reduce the number of cache misses.
Parallel Block 3-D FFT Algorithm

$n_2 n_3$

$P_0 \quad P_1 \quad P_2 \quad P_3$

Partial Transpose

$n_B$

All-to-All comm.

$P_0 \quad P_1 \quad P_2 \quad P_3$

Partial Transpose

$n_2 n_1$

$P_0 \quad P_1 \quad P_2 \quad P_3$

Partial Transpose

$n_1$

$n_2 n_3$
Performance Results

• To evaluate the implemented parallel 3-D FFTs, we compared
  – The implemented parallel 3-D FFT, named FFTE (ver 3.2, supports SSE2, using MPI)
  – FFTW (ver. 2.1.5, not support SSE2, using MPI)

• Target parallel machine:
  – A 16-node dual PC SMP cluster (dual Xeon 2.8GHz, 2GB DDR-SDRAM / node, Linux 2.4.20smp).
  – Interconnected through a Myrinet-2000 switch.
  – MPI-SCore [http://www.pccluster.org] was used.
  – We used an intra-node MPI library for the PC SMP cluster.
Performance of parallel 3-D FFTs
(dual Xeon PC SMP cluster, N1 x N2 x N3 = 2^24 x P)

Number ofNodes

 MFLOPS

0 1000 2000 3000 4000 5000

1 2 4 8 16

FFTE (SSE2, 1CPU)
FFTE (SSE2, 2CPUs)
FFTE (x87, 1CPU)
FFTE (x87, 2CPUs)
FFTW (1CPU)
FFTW (2CPUs)
Breakdown of parallel 3-D FFTs
(dual Xeon PC SMP Cluster, N1xN2xN3=2^24xP)

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x1</td>
<td>0</td>
</tr>
<tr>
<td>1x2</td>
<td>1</td>
</tr>
<tr>
<td>2x1</td>
<td>2</td>
</tr>
<tr>
<td>2x2</td>
<td>3</td>
</tr>
<tr>
<td>4x1</td>
<td>4</td>
</tr>
<tr>
<td>4x2</td>
<td>5</td>
</tr>
<tr>
<td>8x1</td>
<td>6</td>
</tr>
<tr>
<td>8x2</td>
<td>7</td>
</tr>
<tr>
<td>16x1</td>
<td>8</td>
</tr>
<tr>
<td>16x2</td>
<td>9</td>
</tr>
</tbody>
</table>

Green: Comp.  
Blue: Comm.
Parallel Implementation of Classical Gram-Schmidt Orthogonalization

• Gram-Schmidt orthogonalization process is one of the fundamental algorithms in linear algebra.
  – Density-functional theory (DFT) code includes Gram-Schmidt orthogonalization of a large set of wave functions.

• 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

• Modified Gram-Schmidt (MGS) algorithm
  – Cannot be expressed by Level-2 BLAS.
  – Parallel implementation requires additional communication.

• Classical Gram-Schmidt (CGS) algorithm
  – 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.
Classical Gram-Schmidt Algorithm

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

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

- \(a_j\) Raw data vector
- \(q_j\) Orthogonalized vector
- \(\|q_j\|\) Euclid norm
- \((q_i, a_j)\) Inner product
Naive Implementation

- CGS for $m$ by $n$ matrix can be computed by
  - $(m - 1)$ matrix-vector multiplications (GEMV: Level-2 BLAS)
  - $m$ normalizations (NRM2 and SCAL: Level-1 BLAS)

Let the matrix $A$ is denoted as $A = (a_1 a_2 \cdots a_n)$. 

$q_1 = a_1$, $q_1 = q_1 / \|q_1\|$

$q_2 = a_2 - (q_1, a_2)q_1$, $q_2 = q_2 / \|q_2\|$

$q_3 = a_3 - (q_1, a_3)q_1 - (q_2, a_3)q_2$, $q_3 = q_3 / \|q_3\|$

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

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

$q_6 = a_6 - (q_1, a_6)q_1 - (q_2, a_6)q_2 - (q_3, a_6)q_3 - (q_4, a_6)q_4 - (q_5, a_6)q_5$, $q_6 = q_6 / \|q_6\|$
CGS using Matrix Multiplication

• The CGS orthogonalization of a matrix can be changed into a matrix multiplication (GEMM: Level-3 BLAS).

\[ q_1 = a_1, \quad q_1' = \frac{q_1}{\|q_1\|} \]
\[ q_2 = a_2 - (q_1, a_2)q_1, \quad q_2' = \frac{q_2}{\|q_2\|} \]
\[ q_3 = a_3 - (q_1, a_3)q_1 - (q_2, a_3)q_2, \quad q_3' = \frac{q_3}{\|q_3\|} \]
\[ q_4 = a_4 - (q_1, a_4)q_1 - (q_2, a_4)q_2 - (q_3, a_4)q_3, \quad q_4' = \frac{q_4}{\|q_4\|} \]
\[ q_5 = a_5 - (q_1, a_5)q_1 - (q_2, a_5)q_2 - (q_3, a_5)q_3 - (q_4, a_5)q_4, \quad q_5' = \frac{q_5}{\|q_5\|} \]
\[ q_6 = a_6 - (q_1, a_6)q_1 - (q_2, a_6)q_2 - (q_3, a_6)q_3 - (q_4, a_6)q_4 - (q_5, a_6)q_5, \quad q_6' = \frac{q_6}{\|q_6\|} \]
Column-wise Blocking CGS

begin CBCGS($A, Q, N, M, s, h$)
do $j = s, h, M$

$q_j = q_j / \|q_j\|$
do $i = j + 1, j + M$

$w = Q_{s,s+i}^T a_{s+i}$
$q_{s+i} = Q_{s-i,s} w$
$q_{s+i} = q_{s+i} / \|q_{s+i}\|$
end do

$W = Q_{j+M,N-(j+M)}^T A_{j,j+M}$
$Q_{j+M,N-(j+M)} = WQ_{j+M,N-(j+M)}$
end do
end
Recursive Blocking CGS

\[
\text{begin RBCGS}(A, Q, N, M, s, h)
\]

\[
\text{if } (h \leq M) \text{ then}
\]

\[
q_s = q_s / \|q_s\|
\]

\[
do i = 1, h
\]

\[
w = Q^{T}_{s,s+i} a_{s+i}
\]

\[
q_{s+i} = Q_{s-i,s} w
\]

\[
q_{s+i} = q_{s+i} / \|q_{s+i}\|
\]

\[
do\end do
\]

\[
\text{else}
\]

\[
\text{RBCGS}(A, Q, N, M, s, h/2)
\]

\[
W = Q^{T}_{s-(h/2),s+(h/2)} A_{s-(h/2),s+(h/2)}
\]

\[
Q_{s-(h/2),s+(h/2)} = W Q_{s-(h/2),s+(h/2)}
\]

\[
\text{RBCGS}(A, Q, N, M, s+h/2, h/2)
\]

\[
de\end if\end end
\]

Using Matrix-Vector Multiplication (GEMV)

Using Matrix-Matrix Multiplication (GEMM)
Evaluation Environment

• A 32-node Xeon PC-cluster
  – Xeon 3GHz, 1GB DDR2 Memory
• 1000Base-T Gigabit Ethernet
• OpenMPI 1.2.3
• GotoBLAS r1.19
• Intel C Compiler 10.0
• Linux 2.6.20-1.2933.fc6smp
• All programs were run in 64-bit mode
Performance on Xeon 3GHz (1CPU)

Matrix Size vs. GFLOPS for Naive, Column-wise CGS, and Recursive CGS methods.
Performance on 32 node 3GHz Xeon PC Cluster

- Naive
- Column-wise CGS
- Recursive CGS