Recursive inverse factorization

Anton Artemov

Division of Scientific Computing, Uppsala University

anton.artemov@it.uu.se

06.09.2017
Context of work

Research group on large scale electronic structure computations at TDB, UU:

- **Ergo** - a quantum chemistry program for large-scale self-consistent field calculations. http://ergoscf.org
- Parallel programming models
- **Chunks and Tasks** model and library http://chunks-and-tasks.org

Introduction

- Electron density matrix $D$ for a given Fock or Kohn–Sham matrix $F$:

$$D = \sum_{i=1}^{\text{nocc}} c_i c_i^T,$$

(1)

where $c_i$ are eigenvector solutions of the generalized eigenvalue problem:

$$Fc_i = \lambda_i Sc_i, \ i = 1, \ldots, n,$$

(2)

where $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_{\text{nocc}} \leq \lambda_{\text{nocc}+1} \leq \ldots \leq \lambda_n$, $\text{nocc}$ is the number of occupied electron orbitals and $S$ is the basis set overlap matrix.

- Standard way to compute $D$ is via complete solution of eigenvalue problem.
Simple and efficient method is to see the problem as a matrix function

\[ D = \Theta (\mu I - F), \]  

(3)

where \( \Theta \) is Heaviside function, \( \mu \in ]\lambda_{\text{noc}} , \lambda_{\text{noc}+1} [ \) and to use polynomial expansion of \( F \).

Sparse matrix data structures give \( O(n) \) complexity;

expansion can be performed directly in \( F \);

more efficient to perform congruence transformation:

\[ S \rightarrow Z^T SZ = I, \quad F \rightarrow Z^T FZ = \hat{F}. \]  

(4)

and transform eigenvalue problem (2) into standard form:

\[ \hat{F} y = \lambda y, \quad Zy = x, \quad S^{-1} = ZZ^T. \]  

(5)
Motivation

- Common techniques for inverse factorization:
  - Löwdin
  - Inverse Cholesky

provide decay of matrix elements.

- Generally their complexity is $O(n^3)$.

- Alternatively: iterative refinement, but sufficiently accurate starting guess is necessary.

- Main idea - combine iterative refinement with a binary principal submatrix decomposition as starting guess.
Iterative refinement

- Non-local refinement:
  \[
  Z_{i+1} = Z_i \sum_{k=0}^{m} b_k \delta_i^k, \quad b_k = \frac{2k - 1}{2k}, \quad b_0 = 1, \quad \delta_i = I - Z_i^* S Z_i.
  \]
  \[(6)\]

- Local refinement:
  \[
  \begin{aligned}
  \delta_{i+1} &= \delta_i - Z_{i+1}^* S (Z_{i+1} - Z_i) - (Z_{i+1} - Z_i)^* S Z_i, \\
  Z_{i+1} &= Z_i \sum_{k=0}^{m} b_k \delta_i^k.
  \end{aligned}
  \]
  \[(7)\]
Recursive Inverse Factorization

Algorithm  REC-INV-FACT  

Input:  Hermitian positive definite matrix $S$  

Output:  $Z$  

1: if lowest level then 
2: Factorize $S^{-1} = ZZ^*$ and return $Z$  
3: end if  
4: Matrix partition $S = \begin{bmatrix} A & B \\ B^* & C \end{bmatrix}$  
5: $Z_A = \text{REC-INV-FACT}(A)$, $Z_C = \text{REC-INV-FACT}(C)$  
6: $Z_0 = \begin{bmatrix} Z_A & 0 \\ 0 & Z_C \end{bmatrix}$  
7: $\delta_0 = I - Z_0^*SZ_0$  
8: repeat  
9: $Z_{i+1} = Z_i \sum_{k=0}^m b_k \delta_i^k$  
10: $\delta_{i+1} = I - Z_{i+1}^*SZ_{i+1}$  
11: until $\|\delta_{i+1}\| > \|\delta_i\|^{m+1}$  
12: return $Z_{i+1}$
Dynamical hierarchical algorithms

- Dense matrix-matrix-multiplication is good for parallelization.
- Here, nonzero pattern is not known in advance and may change dynamically.
- This makes parallelization for distributed memory difficult.
Conventional parallelization (MPI)

Application programmer responsibilities:

Conventional parallelization (MPI)
Distribute work
Distribute data
Send and receive messages
### MPI vs Chunks and Tasks

**Application programmer responsibilities:**

<table>
<thead>
<tr>
<th>Conventional parallelization (MPI)</th>
<th>Chunks and Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribute work</td>
<td>Divide work into smaller parts</td>
</tr>
<tr>
<td>Distribute data</td>
<td>Divide data into smaller pieces</td>
</tr>
<tr>
<td>Send and receive messages</td>
<td>Register “tasks” and “chunks”</td>
</tr>
</tbody>
</table>
Chunks and Tasks programming model

- Developed for dynamic hierarchical algorithms
- Abstractions for both data and work (chunks and tasks)
- No explicit communication calls in user code
- Determinism, freedom from race conditions and deadlocks
- Feasible to implement efficient backends
- Fail safety: local recovery possible
Chunks and Tasks programming model

- Developed for dynamic hierarchical algorithms
- Abstractions for both data and work (chunks and tasks)
- No explicit communication calls in user code
- Determinism, freedom from race conditions and deadlocks
- Feasible to implement efficient backends
- Fail safety: local recovery possible

Main concepts:
- **Chunk** is defined by data members, serialization functions, may contain identifiers to other chunks giving rise to chunk hierarchies.
- **Task** is defined by input chunk types, output chunk type, work to be performed.
Computation driven by registration of tasks
Like Cilk, StarPU, SuperGlue, XKaapi but
unlike Charm++ (where the computation is driven by messages)
⇒ No message passing

Recursive nesting of tasks allowed
Like Cilk, SuperGlue, XKaapi
but unlike StarPU
⇒ Scalable, good for hierarchic dynamic algorithms

Abstractions for both work and data
Previous task-based approaches mostly focused on shared memory. Distributed memory: either user provides data distribution or all data is managed by a “master” node
⇒ User does not have to provide data distribution
Related programming models

Identifiers to chunks provided by the library

*Unlike Linda and Concurrent Collections*

⇒ Makes it feasible to make data available efficiently
⇒ Leads to restrictions in how data can be accessed

Chunks are readonly once they are registered

*Unlike Linda but similar to Concurrent Collections*

⇒ Chunk cache coherence not an issue
⇒ Determinism (easier for the user to understand her code)
CHT-MPI library

- Publicly available since May 2014 (modified BSD license)
- Task scheduler based on work stealing
- Scalable: No “master node” managing all work or data

Webpage: www.chunks-and-tasks.org
Ref:[Parallel Computing, 40: 328-343 (2014)]
Chunks and Tasks matrix library

- Sparse quad-tree representation:
  - Different leaf matrix representations can be used.
  - Sparsity is dynamically exploited at all levels by skipping operations on zero submatrices.

Result: close to linear scaling with system size for matrix-matrix multiplication for block-sparse leaf level matrices [Parallel Computing, 57: 87-106 (2016)].
Experimental results

Experimental setup:

- Two kinds of molecules used: Glu-Ala helices (quasi-linear) and three-dimensional water clusters.
- Two standard Gaussian basis sets: STO-3G and the larger 3-21G basis set.
- Parallelization done with CHT-MPI.
- Recursive inverse Cholesky with truncation is employed when reaching certain size.
- Computations were performed on Beskow, a Cray XC40 system with Intel Xeon E5-2698v3 cores.
- Resources provided by Swedish National Infrastructure for Computing (SNIC) through PDC Center for High Performance Computing at the KTH Royal Institute of Technology in Stockholm.
Experimental results

Glu-Ala, STO-3G, scaling with system size

Glu-Ala, STO-3G, NNZ per row
Experimental results

Water clusters, STO-3G, scaling with system size

Wall time, seconds

System size

Water clusters, STO-3G, NNZ per row

Number of non-zeros (NNZ) per row

System size × 10^4

Water clusters, STO-3G, scaling with system size

RIF 2p
RIF 4p
RIF 8p
RIF 16p
Linear
Experimental results

Glu-Ala, 3-21G, scaling with system size

Glu-Ala, 3-21G, NNZ per row
Experimental results

Water clusters, 3-21G, scaling with system size

- **RIF 2p**
- **RIF 4p**
- **RIF 8p**
- **RIF 16p**

Water clusters, 3-21G, NNZ per row

- **S**
- **Z**

Graphs show the Wall time, seconds, and Number of non-zeros (NNZ) per row with varying system sizes.
Experimental results

Glu-Ala, approximate weak scaling

- STO-3g, 84*10^3 basis functions per process
- 3-21G, 153*10^3 basis functions per process

Glu-Ala, NNZ per row

- S STO-3G
- Z STO-3G
- S 3-21G
- Z 3-21G
Experimental results

Water clusters, weak scaling

- STO-3G, 1690 basis functions per process
- 3-21G, 3543 basis functions per process

Water clusters, NNZ per row

- S STO-3G
- Z STO-3G
- S 3-21G
- Z 3-21G
Critical path

- **Critical path** — the total amount of work required by an ideal infinite-processor execution.
- Or the longest sequence of operations, which have to be executed serially due to dependencies.
- Let $\Theta(N)$ be the critical path length (c.p.l.) for matrix recursive inverse factorization, $\xi(N)$ be c.p.l. of matrix-matrix multiplication.

Assumptions:

- Leaf level are of size $bs \Rightarrow L = \log_2(N/bs)$ levels in the hierarchy.
- Matrix sum has c.p.l. $\log_2(N)$, leaf-level routines have c.p.l. of 1.
Critical path

Then

- Iterative refinement has c.p.l. \( C_1 \cdot \xi(N_{\text{cur}}) + C_2 = \mu(N_{\text{cur}}) \).
- Total number of tasks:

\[
\sum_{i=0}^{L-1} 2^i \left( 1 + \mu \left( \frac{N}{2^i} \right) \right) + 2^L. \tag{8}
\]
Critical path

Then

- Iterative refinement has c.p.l. $C_1 \cdot \xi(N_{cur}) + C_2 = \mu(N_{cur})$.
- Total number of tasks:

$$\sum_{i=0}^{L-1} 2^i \left(1 + \mu \left(\frac{N}{2^i}\right)\right) + 2^L.$$  \hspace{1cm} (8)

- All recursive calls are handled in parallel:

$$\Theta(N) = \sum_{i=0}^{L-1} \left(1 + \mu \left(\frac{N}{2^i}\right)\right) + 1.$$  \hspace{1cm} (9)
Critical path

Then

- Iterative refinement has c.p.l. $C_1 \cdot \xi(N_{cur}) + C_2 = \mu(N_{cur})$.
- Total number of tasks:

  \[
  \sum_{i=0}^{L-1} 2^i \left( 1 + \mu \left( \frac{N}{2^i} \right) \right) + 2^L. \tag{8}
  \]

- All recursive calls are handled in parallel:

  \[
  \Theta(N) = \sum_{i=0}^{L-1} \left( 1 + \mu \left( \frac{N}{2^i} \right) \right) + 1. \tag{9}
  \]

- Assuming $\xi(N_{cur}) = \log_2^2(N_{cur})$:

  \[
  \Theta(N) \leq O(\log^3_2(N)) \tag{10}
  \]
Experimental results

Least-squares fitting shows that c.p.l. grows like $O(\log_2 N)$ rather than $O(\log_3 N)$. 

**Glu-Ala molecules, critical path**

- STO-3G
- 3-21G

**Water cluster, critical path**

- STO-3G
- 3-21G
Experimental results

Least-squares fitting shows that c.p.l. grows like $O(\log^2(N))$ rather than $O(\log^3(N))$
Conclusions

Presented algorithm
▶ can be built on top of more demanding algorithm used on leaf-level computations;
▶ scales almost linearly with system size;
▶ has weak scaling better than any power function;
▶ has critical path with length $\leq O(\log^2 N)$.
Conclusions

Presented algorithm

- can be built on top of more demanding algorithm used on leaf-level computations;
- scales almost linearly with system size;
- has weak scaling better than any power function;
- has critical path with length $\leq O(\log_2^3(N))$. 
References


▶ www.chunks-and-tasks.org
The End
Any Questions?