Parallel Computing 40 (2014) 47–58

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Sparse matrix multiplication: The distributed block-compressed sparse row library

Urban Borštnik [a,*,1], Joost VandeVondele [b], Valéry Weber [a,2], Jürg Hutter [a]

[a] Physical Chemistry Institute, University of Zürich, Winterthurerstrasse 190, CH-8057 Zürich, Switzerland
[b] Department of Materials, ETH Zürich, Wolfgang-Pauli-Strasse 27, 8093 Zürich, Switzerland

A R T I C L E   I N F O

Article history:
Received 16 April 2013
Received in revised form 10 March 2014
Accepted 24 March 2014
Available online 1 April 2014

Keywords:
Sparse matrix
Parallel sparse matrix multiplication
Quantum chemistry

A B S T R A C T

Efficient parallel multiplication of sparse matrices is key to enabling many large-scale calculations. This article presents the DBCSR (Distributed Block Compressed Sparse Row) library for scalable sparse matrix–matrix multiplication and its use in the CP2K program for linear-scaling quantum-chemical calculations. The library combines several approaches to implement sparse matrix multiplication in a way that performs well and is demonstrably scalable. Parallel communication has well-defined limits. Data volume decreases with $\mathcal{O}(1/\sqrt{P})$ with increasing process counts P and every process communicates with at most $\mathcal{O}(\sqrt{P})$ others. Local sparse matrix multiplication is handled efficiently using a combination of techniques: blocking elements together in an application-relevant way, an autotuning library for small matrix multiplications, cache-oblivious recursive multiplication, and multithreading. Additionally, on-the-fly filtering not only increases sparsity but also avoids performing calculations that fall below the filtering threshold. We demonstrate and analyze the performance of the DBCSR library and its various scaling behaviors.

1. Introduction

Dense matrix–matrix multiplication is one of the most basic operations in linear algebra. Highly optimized implementations, both serial and parallel, are available and the underlying algorithms are well understood [1–6]. Somewhat surprisingly, the same is not true for sparse matrix–matrix multiplication [7–13]. The reason might be that many problems can be solved based on a sparse matrix vector multiplication kernel. In the field of computational chemistry, physics, and material science an important exception can be found, namely the problem of solving the self consistent field (SCF) equations that arise in Kohn–Sham or Hartree–Fock theory. The solution of the SCF equations is a matrix that minimizes an energy functional subject to constraints, such that the solution matrix is idempotent ($\mathbf{A} \times \mathbf{A} = \mathbf{A}$). Traditionally this solution matrix, named the density matrix, is found using diagonalization techniques. Typically 10–50% of the eigenvectors of the Hamiltonian matrix of the system are needed to build the density matrix. As the system size increases, both the density matrix and the Hamiltonian matrix become sparse but the eigenvectors do not. This is an opportunity to avoid the cubically scaling diagonalization step and to directly compute the density matrix using linear scaling techniques in which the computational

* Corresponding author. Present address: High Performance Computing Group, Information Technology Services, ETH Zürich, SOW H 12, Sonneggstrasse 63, CH-8092 Zürich, Switzerland. Tel.: +41 44 632 3512.
E-mail address: urban.borstnik@id.ethz.ch (U. Borštnik).
1 While on leave from the National Institute of Chemistry, Ljubljana, Slovenia.
2 Present address: IBM Research Division, Zürich Research Laboratory, 8803 Ruschlikon, Switzerland.

http://dx.doi.org/10.1016/j.parco.2014.03.012
0167-8191/© 2014 Elsevier B.V. All rights reserved.

effort scales linearly with an increase in system size. Among the various options, the density matrix can be obtained as a (matrix) function of the Hamiltonian matrix. The Hamiltonian matrix can itself be obtained as a Chebyshev expansion or from recursion relations [12]. In both cases the most important operation is sparse matrix–matrix multiplication. As a result, several groups that develop linear scaling SCF methods have reported on the development of sparse matrix–matrix multiplication algorithms and libraries. [7,10–14].

The sparsity of the Hamiltonian and density matrix depends on the chemistry of the underlying system, the geometric arrangement of the atoms, and the choice of the basis set. For the common case of atom-centered basis sets, the magnitude of the matrix elements decays with increasing distance between the atoms. Typically, for a three-dimensional atomic system, 10,000 s of matrix elements per row are non-negligible compared to a given threshold. Product matrices retain this sparsity ratio. The computational cost for multiplication is therefore large, and parallel computing is thus essential to have a reasonable time to solution. For a wide range of interesting problems that are affordable on current supercomputer hardware, the occupation is high (i.e., the percentage of non-zero elements is thus large), on the order of 10%, but remains so during the procession of multiplications in the calculations. Good performance in the limit of such high occupations should therefore be an important design criterion. A further aspect that has to be considered is the fact that some internal structure is present in the sparsity pattern. In particular, matrix elements are naturally grouped into "atomic blocks" or sub-matrices, which correspond to the interactions between basis functions centered on a given pair of atoms. It is natural and efficient to use this structure to enhance the performance of the implementation.

We present a sparse matrix multiplication library which takes these two points into account and aims to provide good performance for these types of matrices. In particular, we aim for an algorithm that becomes equal to the known optimal algorithms for the dense matrix multiplication in the case of a sparse matrix with 100% occupation. In particular, all-to-all communication is avoided in that case, too. Furthermore, a cache-oblivious strategy is introduced that enhances the FLOP rate, and an autotuned library is developed that performs small matrix multiplications efficiently. Because the library is used not just for matrix multiplication but also other matrix operations as well as a storage container for matrices, the design choices are constrained by these requirements.

In the following section we present the implementation of the DBCSR sparse matrix multiplication library. We describe the data storage layout used, data distribution and transfer among distributed-memory nodes, and the node-local multiplication approach. In the results and discussion section we present the performance of the library for common sparsities and core counts. The performance of an actual application is analyzed in the Application Performance: Linear Scaling Computational Cost section. This section is followed by the Performance Limits in which the performance at the strong and weak scaling limits is reported and analyzed.

2. Sparse matrix multiplication

2.1. Matrix storage

DBCSR (Distributed Blocked Compressed Sparse Row) matrices are stored in blocked compressed sparse row (CSR) format distributed over a two-dimensional grid of processes.

Individual matrix elements are grouped into blocks by rows and columns. Block sizes are chemically motivated, e.g., based on the number of basis functions used for an atom type. The blocked rows and columns form a grid of blocks. It is these blocks that are indexed by the CSR index. Indexing the blocks instead of individual elements makes the index smaller since there are far fewer blocks than individual elements in a matrix for most basis sets. Indexing by blocks also makes lookups by atom number, which is a very common operation, much easier.

The blocks of the matrix are distributed over a rectangular process grid. While an arbitrary rectangular grid can be used, the dimensions of the process grid are chosen so that the least common multiple of the two grid dimensions is minimized, as we later explain. Square grids are preferred because they minimize the number of messages exchanged. The blocked matrix rows and columns can be arbitrarily mapped to the process rows and columns. Two mapping functions are defined: $p_r(r)$ maps block rows r to process rows $p_r(r)$ and $p_c(c)$ maps block columns c to process columns $p_c(c)$. Any matrix block (r,c) is therefore assigned to a process $(p_r(r),p_c(c))$ from the process grid. To prevent load imbalance it is best to provide a mapping in which rows of equal block size are evenly distributed among the process rows and columns. In practice, the randomization is performed once at the beginning of a program and is used for all subsequent matrix operations, including matrix multiplication. A sample permutation of a diagonal-heavy sparse matrix is shown in Fig. 1. In addition it is beneficial for the row and column mappings to be similar. The library provides a routine to help match the row and column mappings for non-square process grids.

Symmetric matrices, including antisymmetric, Hermitian, and antihermitian matrices, are stored with just one half of the matrix. In these matrices only the upper triangle is stored, i.e., blocks $(r,c),\ c \geqslant r$. To keep data balanced among the processes, these blocks are not all stored on the process row and column corresponding directly to the row and column distribution $(p_r(r),p_c(c))$. Such a mapping would be grossly imbalanced. Instead blocks for which $r+c$ is divisible by 2 are stored at the transposed process in process row and column $(p_r(c),p_c(r))$.