Redesign of Higher-level Matrix Algorithms for Multicore and Distributed Architectures and Applications in Quantum Monte Carlo Simulation

Che-Rung Lee
Department of Computer Science
National TsingHua University
Hsinchu, Taiwan 30013
cherung@cs.nthu.edu.tw

Zhaojun Bai
Department of Computer Science
University of California
Davis, CA 95616, USA
bai@cs.ucdavis.edu

***Abstract*—Numerical algorithm runtimes are increasingly dominated by the cost of communication (memory access), which can exceed the cost of floating point operations by orders of magnitude. A great deal of efforts had been focused on the design of communication-avoiding parallel matrix operations using techniques such as blocking or tiling. However, not all matrix operations can be efficiently parallelized by these techniques. A matrix operation is referred to as a *hard-to-parallel matrix operation* (HPMO) if there have hard serial bottlenecks that are hardly parallelizable. Otherwise, it is referred to as an *easy-to-parallel matrix operation* (EPMO). The performance scalability of an HPMO is significantly poorer than an EPMO on multicore and distributed architectures. The design of an application higher-level algorithm should try to avoid the use of HPMOs as computational kernels.**

In this paper, as a case study, we present an HPMO-avoiding algorithm for the Green's function calculation in quantum Monte Carlo simulation. The original algorithm utilizes the QR-decomposition with column pivoting (QRP) as computational kernel. QRP is an HPMO. The redesigned algorithm maintains the same simulation stability but employs the standard QR decomposition without pivoting (QR), which is an EPMO. Different implementations of the redesigned algorithm on multicore and distributed architectures are investigated. Although some implementations of the redesigned method use about a factor of three more floating-point operations than the original algorithm, they are about 20% faster on a quad-core system and 2.5 times faster on a 1024-CPU distributed system. The broader impact of the redesign of higher-level matrix algorithms to avoid HPMOs in other computational science applications will also be discussed.

***Index Terms*—Matrix algorithm; Quantum Monte Carlo simulation; Multicore algorithm; Distributed algorithms; Communication avoid algorithm; HPMO and EPMO.**

I. Introduction

We are experiencing a dramatic transformation of computing landscape to multicore and distributed systems. The shift to an increasing number of cores and heterogeneous architectures requires significant modification to today's computational tools and technologies. The communication cost of an algorithm has already exceeded arithmetic cost by orders of magnitude, and the gap is growing exponentially over time [9]. To compensate the speed gap, new principles of software and algorithm design for performance are gradually built via extensive studies. For instance, one may use massive fine grained and asynchronous threading to improve processor utilization and to hide the communication latency [7], or trade with additional computation for reducing communication among processing units [8].

Those ideas have been widely applied in designing basic matrix operations and algorithms. In [1], the pivoting in the LU decomposition is avoided by using randomization technique and the QR decomposition. In [17], [7], [6], [15], tiled algorithms with dynamically scheduling are used to achieve fine granularity and asynchronicity for Cholesky, LU and QR factorizations and Hessenberg reduction. In [20], [8], processor utilizations are enhanced by minimizing communication cost on various platforms.

However, not all matrix operation can be efficiently parallelized for great efficiency. For example, the QR decomposition with column pivoting (QRP) performs significantly poorer than the QR without pivoting (see Figure II.1). One challenge is that the pivoting involves moving the columns of the matrix between levels of a memory hierarchy and/or between processors over a network. Another challenge is that the pivoting criterion is computed based on the global runtime information, which requires a global synchronization when subtasks run in parallel. The first challenge alone may not be a critical performance killer, since it can be improved by techniques like pre-fetching and computation and communication overlapping. It is the combinative effects of those two challenges that jeopardize an effective performance improvement by parallelization.

We call a matrix operation such as the pivoted QR a *hard-to-parallel matrix operation* (HPMO). On the other hand, matrix operation such as matrix-matrix multiplication and the QR decomposition (without pivoting) are called *easy-to-parallel matrix operations* (EPMOs). By nature, it is not likely to improve the performance of an HPMO significantly by parallelization. In [1], the LU decomposition with pivoting for solving a linear system of equations and matrix inversion is recommended to be replaced by the QR decomposition.

For large-scale application simulations that currently employ HPMOs as computational kernels, it is highly desired and often challenging to redesign the kernel to avoid the use of HPMOs. In this paper, we present an HPMO-avoiding algorithm for a real physical simulation application: the Green's function in Determinant Quantum Monte Carlo (DQMC) simulation [2], [14]. For stabilizing the matrix multiplication and inversion, the original algorithm uses the pivoted QR decomposition to stratify matrix elements of different magnitude order [13], [12], [4]. Our redesigned algorithm, called *Structural Orthogonal Factorization* (SOF), reformulates the calculation to avoid the pivoted QR, and uses matrix-matrix multiplication and QR-decomposition only, which both are EPMOs. We will present different implementations of SOF that use BLAS and LAPACK on multicore and PBLAS and ScaLAPACK on distributed architecture as building blocks to harvest the great performance of those highly optimized numerical libraries. Experiment results showed the SOF algorithm is as stable as the original one. Moreover, although the operation count of the SOF algorithm is three times more than that of the original algorithm, the SOF still outperforms the original algorithm by 20% on a quad-core system, and 250% on a 1024-CPU distributed system.

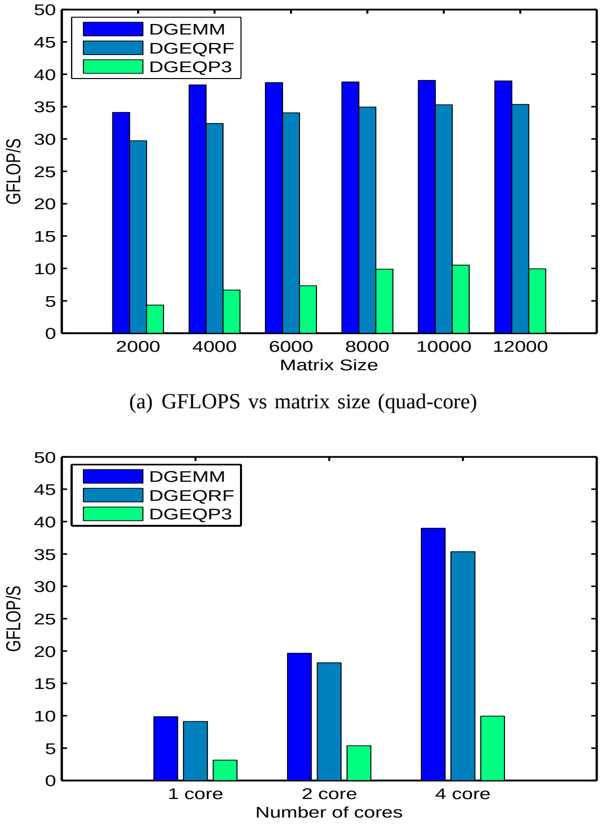
The rest of the paper is organized as follows. Section II compares the performance and scalability of three matrix operations on multicore and distributed architectures, namely matrix-matrix multiplication, QR, and pivoted QR. Based on their properties of parallelization, they are classified as HPMOs or EPMOs. Section III introduces the Green's function calculation in the determinant quantum Monte Carlo simulations, and the HPMO-based algorithm. The redesigned algorithm, structured orthogonal factorization (SOF), is presented in section IV. Section V illustrates different implementations of SOF utilizing highly optimized numerical libraries and analyzes their operation counts. Section VI compares the performance of various implementations of SOF to the original algorithm. The concluding remarks and future work are given in section VII.

II. EPMO AND HPMO

This section begins with the performance and scalability measurement of some basic matrix operations from well-established BLAS and LAPACK on multicore and PBLAS and ScaLAPACK distributed architectures. From there, a classification of easy-to-parallel matrix operations (EPMOs) and hard-to-parallel matrix operations (HPMOs) is presented.

The following three basic matrix operations are selected for their distinct characteristic of parallelism and relevance to the application studied in this paper.

- The matrix-matrix multiplication $C = AB$ is often used as a performance benchmark to indicate the achievable peak performance on various architectures, where A, B and C are $n \times n$ matrices. Its computational subroutine is DGEMM in BLAS and PDGEMM in PBLAS.
- The QR decomposition is defined as $A = QR$, where Q is an orthogonal matrix and R is an upper triangle matrix. Its computational subroutine is DGEQRF in LAPACK and PDGEQRF in ScaLAPACK.
- The pivoted QR decomposition $AP = QR$, where Q is orthogonal, R is upper triangular and P is a permutation matrix. Its computational subroutine is DGEQP3 in LAPACK and PDGEQP3 in ScaLAPACK.



(a) GFLOPS vs matrix size (quad-core)

(b) GFLOPS vs number of cores

Fig. II.1. Performance of MKL10.2 DGEMM, DGEQRF and DGEQP3 on an Intel i7 Quad 2.66GHz.

Figure II.1 shows the performance of these three basic matrix operations on an Intel Core i7 Quad 2.66GHz machine as available in MKL version 10.2. Figure 1(a) compares the Gflop/s of three subroutines for different matrix size using 4 cores; Figure 1(b) displays their Gflop/s using 1 core, 2 cores, and 4 cores for matrix size 12000. As can be seen, the performance of QR is close to that of the matrix-matrix multiplication. Although the QR and the pivoted QR are kin to each other, the performance of the pivoted QR is much poorer than QR. Furthermore, we notice that the speedup of DGEQP3 on quad core is only a slight better than two-core, as shown in Figure 1(b). By contrast, the performance of DGEMM and DGEQRF are almost quadrupled from one to quad cores. It is anticipated with more cores, the performance improvement of DGEQP3 would be further deteriorate.

To measure the performance of these matrix operations on a message passing based cluster, we used a Cray XT4 massively parallel processing system (named Franklin) at NERSC (Na-

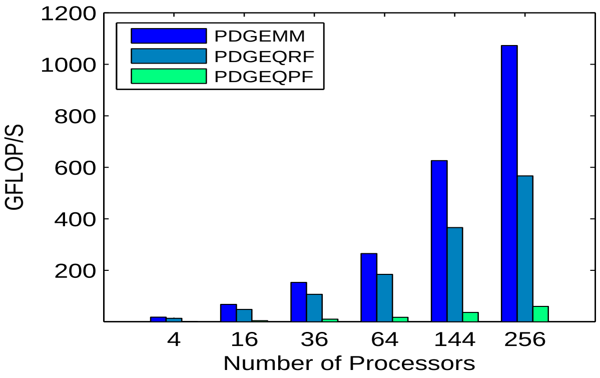


Fig. II.2. Performance of PDGEMM, PDGEQRF and PDGEQP3 on a Cray XT4 at NERSC.

tional Energy Research Scientific Computing Center). Franklin has 9,572 compute nodes. Each compute node consists of a 2.3GHz single socket quad-core AMD Opteron processor (Budapest) with a theoretical peak performance of 9.2 GFlop/s per core (4 flops/cycle if using SSE128 instructions). MPI task assignment follows the SMP placement style. For instance, 8 MPI tasks are distributed on 2 compute nodes as follows

Node	Node 1				Node 2			
Core	1	2	3	4	1	2	3	4
MPI Rank	0	1	2	3	4	5	6	7

To match the nature of matrix algorithms, PBLAS and ScaLAPACK maps processors onto a two-dimensional rectangular grid (process grid) according to the row-major order or the column major order. In our test cases, the square process grid is used, and the number of allocated processors is represented by the size of the grid. For instance, a 2×2 processor allocation means 4 processors are mapped onto a two by two process grid.

We chose a typical matrix size, $16384 = 2^{14}$, to compare the performance differences of these three routines, and used different number of processors, $2\times 2, 4\times 4, 6\times 6, 8\times 8, 12\times 12$ and 16×16 to illustrate their scalability. Figure II.2 shows the performance results of three PBLAS and ScaLAPACK subroutines with respect to different number of processors. It can be seen that the performance of the pivoted QR is only about one tenth of the other two matrix operations and the scalability is extremely poor as well.

There are issues which make the pivoted QR extremely hard to parallelize. First, the pivoting involves exchanging columns of the matrix, which may locate at different levels of a memory hierarchy and/or between processors over a network. Second, the pivoting criterion requires a global synchronization to evaluate the norms of remaining columns. The first issue cannot be resolved by pre-fetching or computation and communication overlapping because of the second issue. Thus, the pivoting, which entails expensive data communication, can be regarded as the non-parallelizable operations of the pivoted QR.

The ineffectiveness of parallelizing the pivoted QR can also be explained by Amdahl's law. Let T_p be the time for the communication requested by pivoting, and T_c be the time for computation. If we assume the communication cannot be parallelized at all and the computation part can be perfectly parallelized, the asymptotical speedup of the parallel pivoted QR is

$$\rho = \frac{T_p + T_c}{T_p}.$$

If T_p is much larger than T_c, the limited speedup and scalability can be anticipated by parallelization.

A number of matrix operations share the same characters as the pivoted QR. We call a matrix operation a *hard-to-parallel matrix operation* (HPMO) if its most time consuming subtasks, including communication, cannot be parallelized. Otherwise, a matrix operation is called a *easy-to-parallel matrix operation* (EPMO). It should note that our classification does not imply that the performance of HPMOs cannot be improved by other methods. For instance, one can improve the performance of pivoted QR by minimizing the communication or by relaxing the pivoting criterion. This is the effort beyond the scope of our current work.

III. Green's function calculation

Green's function calculation is the kernel in quantum Monte Carlo simulations of interacting electrons in computational material science [5], [13], [12]. Specifically, in the Determinant Quantum Monte Carlo (DQMC) simulation, the Green's function is formulated as

$$G = (I + B_L B_{L-1} \cdots B_1)^{-1}, \qquad \text{(III.1)}$$

where $B_i = BV_i$, is a product of a symmetric matrix B and a diagonal matrix V_i [2]. Matrix $B = e^{\Delta\tau K}$ is a matrix exponential of the *hopping matrix* K, which describes how electrons hop among sites. Scalar $\Delta\tau$ is the time discretization parameter. Its product with L, $\beta = \Delta\tau L$, is the inverse temperature. The diagonal elements of matrix V_i are in $\{e^{\lambda}, e^{-\lambda}\}$, where $\lambda = \cosh^{-1}(e^{U\Delta\tau/2})$. Scalar U is an energy parameter that related to local repulsion between electrons [2].

When L is large (a.k.a. low temperatures), the matrix G in (III.1) is extremely ill-conditioned. Several methods have been proposed [13], [12], [18], [4] to stabilize the computation by stratifying the magnitude of elements in the matrix multiplications. All those methods inevitably require the pivoted QR decomposition. Algorithm 1 by Loh *et al* [13], [12] is currently used to calculate the Green's function G. In the algorithm, elements of different energy levels, which correspond to different magnitude of numbers, are stratified by the pivoted QR decomposition. The stratification step protects small numbers to be rounded by mixing with large ones. The stability analysis of the stratification method can be found in [4].

IV. SOF Algorithm

To avoid HPMO QR decomposition with pivoting, Algorithm 2 is a redesigned method to compute the Green's

Algorithm 1 Stratification method

```
1) Compute the pivoted QR: $B_1 = Q_1 R_1 P_1^T$
2) Set $U_1 = Q_1$, $D_1 = \mathrm{diag}(R_1)$ and
   $T_1 = D_1^{-1} R_1 P_1^T$
3) For $i = 2, 3, \cdots, L$
     a) Compute $C_i = (B_i U_{i-1}) D_{i-1}$
     b) Compute the pivoted QR: $C_i = Q_i R_i P_i^T$
     c) Set $U_i = Q_i$, $D_i = \mathrm{diag}(R_i)$, and
        $T_i = (D_i^{-1} R_i)(P_i^T T_{i-1})$
4) Compute $G = T_L^{-1}(U_L^T T_L^{-1} + D_L)^{-1} U_L^T$
```

function G. It is referred to as a SOF (Structured Orthogonal Factorization) method since the computational kernel involves an orthogonal decomposition of a structured matrix. The similar computational kernel has also appeared in other higher-level matrix algorithms, such as inverse-free method for computing the generalized Schur decomposition of a matrix pair [3] and the matrix polar decomposition [11, Chap.8][16].

Algorithm 2 SOF method

```
1) Set $M_2 = I$, $A_2 = B_1$
2) For $i = 2, 3, \cdots, L$
     a) Compute QR decomposition of the matrix
        $$\begin{bmatrix} M_i \\ -B_i \end{bmatrix} = \begin{bmatrix} Q_{11}^{(i)} & Q_{12}^{(i)} \\ Q_{21}^{(i)} & Q_{22}^{(i)} \end{bmatrix} \begin{bmatrix} R_i \\ 0 \end{bmatrix} \qquad \text{(IV.2)}$$
     b) Update $A_{i+1} = (Q_{12}^{(i)})^T A_i$ and
        set $M_{i+1} = (Q_{22}^{(i)})^T$
3) Compute $G = (M_L + A_L)^{-1} M_L$
```

The number of floating point operations of the SOF method is about 3 times to that of the stratification method. However, because the SOF algorithm does not use any HPMOs, its performance can be efficiently improved by parallelization, which will be shown in Section VI.

The correctness of the SOF algorithm is justified as the follows.

Lemma 4.1: Submatrices $Q_{11}^{(i)}$, $Q_{12}^{(i)}$, $Q_{21}^{(i)}$, and $Q_{22}^{(i)}$ are nonsingular for $i = 2, \cdots, L$.

Proof: By the CS decomposition [19, page74-75], $Q_{11}^{(i)}$ and $Q_{22}^{(i)}$ have the same singular values; and so do $Q_{21}^{(i)}$ and $Q_{12}^{(i)}$. Thus we only need to prove the non-singularity of $Q_{11}^{(i)}$ and $Q_{21}^{(i)}$. From (IV.2), we have

$$\begin{cases} M_i &= Q_{11}^{(i)} R_i \\ -B_i &= Q_{21}^{(i)} R_i \end{cases}$$

for $i \geq 2$. Since B_is are nonsingular, $Q_{21}^{(i)}$ and R_i are nonsingular. The non-singularity of $Q_{11}^{(i)}$ is proved by induction. In the base case, $M_2 = I$, which makes $Q_{11}^{(2)}$ nonsingular. Suppose $Q_{11}^{(i-1)}$ is nonsingular. Then $M_i = (Q_{22}^{(i-1)})^T = Q_{11}^{(i)} R_i$ is also nonsingular. Therefore, $Q_{11}^{(i)}$ is nonsingular. ■

Theorem 4.2: $G = (M_L + A_L)^{-1} M_L$

Proof: For $i = 2, 3, \ldots, L$, by premultiplying Q_i^T to (IV.2), we have

$$\begin{bmatrix} (Q_{11}^{(i)})^T & (Q_{21}^{(i)})^T \\ (Q_{12}^{(i)})^T & (Q_{22}^{(i)})^T \end{bmatrix} \begin{bmatrix} M_i \\ -B_i \end{bmatrix} = \begin{bmatrix} R_i \\ 0 \end{bmatrix}.$$

The second block row gives the equations

$$\begin{cases} (Q_{12}^{(2)})^T M_2 - (Q_{22}^{(2)})^T B_2 &= 0 \\ (Q_{12}^{(3)})^T M_3 - (Q_{22}^{(3)})^T B_3 &= 0 \\ &\vdots \\ (Q_{12}^{(L)})^T M_L - (Q_{22}^{(L)})^T B_L &= 0 \end{cases}$$

With $M_2 = I$ and $M_i = (Q_{22}^{(i-1)})^T$ for $i > 2$, the above equations can be rewritten as

$$\begin{cases} (Q_{12}^{(2)})^T &= (Q_{22}^{(2)})^T B_2 \\ (Q_{12}^{(3)})^T (Q_{22}^{(2)})^T &= (Q_{22}^{(3)})^T B_3 \\ &\vdots \\ (Q_{12}^{(L)})^T (Q_{22}^{(L-1)})^T &= (Q_{22}^{(L)})^T B_L \end{cases}$$

By Lemma 4.1, $Q_{22}^{(i)}$ is invertible. Thus,

$$\begin{cases} (Q_{12}^{(2)})^T &= (Q_{22}^{(2)})^T B_2 \\ (Q_{12}^{(3)})^T &= (Q_{22}^{(3)})^T B_3 (Q_{22}^{(2)})^{-T} \\ &\vdots \\ (Q_{12}^{(L)})^T &= (Q_{22}^{(L)})^T B_L (Q_{22}^{(L-1)})^{-T} \end{cases}$$

If multiplying all terms of the above equations from bottom to top, we have

$$\begin{aligned} &(Q_{12}^{(L)})^T \cdots (Q_{12}^{(3)})^T (Q_{12}^{(2)})^T \\ =&(Q_{22}^{(L)})^T B_L (Q_{22}^{(L-1)})^{-T} \cdots \\ &(Q_{22}^{(3)})^T B_3 (Q_{22}^{(2)})^{-T} \cdot (Q_{22}^{(2)})^T B_2 \\ =&(Q_{22}^{(L)})^T B_L \cdots B_3 B_2. \end{aligned} \qquad \text{(IV.3)}$$

From Step 1 and Step 2(b),

$$A_L = (Q_{12}^{(L)})^T \cdots (Q_{12}^{(3)})^T (Q_{12}^{(2)})^T B_1. \qquad \text{(IV.4)}$$

By combining (IV.4) and (IV.3), A_L can be expressed as

$$A_L = (Q_{22}^{(L)})^T B_L \cdots B_3 B_2 B_1.$$

As the result,

$$\begin{aligned} G &= (M_L + A_L)^{-1} M_L \\ &= \left[(Q_{22}^{(L)})^T + (Q_{22}^{(L)})^T B_L \cdots B_2 B_1 \right]^{-1} (Q_{22}^{(L)})^T \\ &= (I + B_L \cdots B_2 B_1)^{-1}. \end{aligned}$$

■

V. Implementations

The most time consuming task in the SOF algorithm is step 2)-a). In this section, we consider a variety of implementations of step 2)–a) on multicore and distributed architectures. We try to use the subroutines of BLAS and LAPACK or PBLAS and ScaLAPACK as much as possible to harvest the performance of those highly performance optimized numerical libraries.

A. On multicore architectures

A straightforward implementation of SOF step 2)-a) to apply the QR decomposition routines as available in LAPACK is shown in Algorithm 3. The subroutine `DGEQRF` computes the QR decomposition while stores the Q-factor as a product of elementary Householder reflectors. Subsequently, the subroutine `DORGQR` is used to form the Q-factor explicitly. The right half of the Q-factor is then extracted. The combination of Algorithms 2 and 3 is referred to as **SOF-M1**.

Algorithm 3 SOF step 2)-a) – multicore version 1

```
1) Compute the QR decomposition by DGEQRF.
2) Form the full $Q$-factor by calling DORGQR.
3) Extract subblocks $Q_{12}^{(\ell)}$ and $Q_{22}^{(\ell)}$.
```

We can improve Algorithm 3 by computing the right half of the Q-factor only. Suppose the Householder elementary reflectors are stored in a $2n \times n$ matrix V. the Q-factor can be written as

$$Q = I - VTV^T, \tag{V.5}$$

where T is an $n \times n$ upper triangular matrix. If the matrix V is conformally partitioned as

$$V = \begin{bmatrix} V_u \\ V_d \end{bmatrix}, \tag{V.6}$$

then the right half of the Q-factor is given by

$$\begin{bmatrix} Q_{12}^{(\ell)} \\ Q_{22}^{(\ell)} \end{bmatrix} = \begin{bmatrix} -V_u T V_d^T \\ I - V_d T V_d^T \end{bmatrix}.$$

Furthermore, note that V_u is a lower triangular matrix, which can be exploited to further reduce the computational cost of the matrix multiplication. This leads to the second implementation of SOF step 2)-a) shown in Algorithm 4. The combination of Algorithms 2 and 4 is referred to as **SOF-M2**.

Algorithm 4 SOF step 2)-a) – multicore version 2

```
1) Perform the QR decomposition by DGEQRF.
2) Compute the $T$ matrix by DLARFT
3) Form $(Q_{12}^{(\ell)})^T = -V_d T^T V_u^T$, and
   $(Q_{22}^{(\ell)})^T = I - V_d T^T V_d^T$ using DGEMM and DTRMM
```

One can use the partial results computed inside `DGEQRF` for form the T matrix, which can save some floating point operation than calling `DLARFT` after `DGEQRF` as in Algorithm 4. If we represent T in the block format,

$$T = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1b} \\ & T_{22} & & T_{2b} \\ & & \ddots & \vdots \\ & & & T_{bb} \end{bmatrix},$$

Subroutine `DGEQRF` produces T_{ii} in block. The computation of T can take advantage of those partial results. Suppose in the middle of `DGEQRF`, the partial Householder transformation is $H_1 = I - V_1 T_1 V_1^T$, and $H_2 = I - V_2 T_2 V_2^T$ is the next generated block Householder transformation. The merging of H_1 and H_2 is

$$\begin{aligned} H_1 H_2 &= (I - V_1 T_1 V_1^T)(I - V_2 T_2 V_2^T) \\ &= I - \begin{bmatrix} V_1 & V_2 \end{bmatrix} \begin{bmatrix} T_1 & -T_1 V_1^T V_2 T_2 \\ 0 & T_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix}, \end{aligned}$$

which shows the T matrix of merged Householder transformation is

$$\begin{bmatrix} T_1 & -T_1 V_1^T V_2 T_2 \\ 0 & T_2 \end{bmatrix}. \tag{V.7}$$

Thus, with additional computation of $-T_1 V_1^T V_2 T_2$ for each block, the entire T matrix can be computed recursively. This leads to the third implementation of SOF step 2)-a), sketched in Algorithm 5. The combination of Algorithms 2 and 5 is referred to as **SOF-M3**.

Algorithm 5 SOF step 2)-a) – multicore version 3

```
1) Modify DGEQRF to generate additional $T$ matrix, as
   described in (V.7).
2) Form $(Q_{12}^{(\ell)})^T = -V_d T^T V_u^T$ and
   $(Q_{22}^{(\ell)})^T = I - V_d T^T V_d^T$ using DGEMM and DTRMM
```

SOF-M1 is easy to implement but performs most unnecessary computations. SOF-M3 has the least computational cost but involves those routines that are not optimized for targeted platforms. SOF-M2 is in between, which can be implemented by invoking routines from BLAS and LAPACK libraries. The leading operation counts and the number of calls of the BLAS and LAPACK subroutines used in the stratification method and SOF methods are summarized in Tables V.1.

B. On massive parallel processing architectures

Two implementations of SOF method step 2)-a) on distributed architectures are shown in Algorithms 6 and 7. They are similar to Algorithms 3 and Algorithm 5 respectively, except using the corresponding subroutines from PBLAS and ScaLAPACK. The lack of the corresponding version of Algorithm 4 is because the `PDLARFT` subroutine does not generate global T matrix. The combinations of Algorithm 2 with Algorithms 6 and 7 will be denoted as **SOF-H1** and **SOF-H2**, respectively.

Algorithm 6 SOF step 2)-a) – distributed version 1

```
1) Perform the QR decomposition by using PDGEQRF.
2) Form the full $Q$-factor by calling PDORGQR.
3) Extract the subblocks $Q_{12}^{(\ell)}$ and $Q_{22}^{(\ell)}$ from the $Q$-factor.
```

VI. Experiments

We begin with a validation of numerical stability and accuracy of the SOF method by comparing with the stratification method and then report the performance of the stratification and SOF methods on multicore and distributed architectures.

TABLE V.1
OPERATION COUNTS AND NUMBER OF CALLS OF BLAS AND LAPACK SUBROUTINES BY STRATIFICATION AND SOF METHODS

		Op.counts	No. of calls.			
Subroutine	Function		strati.alg	SOF-M1	SOF-M2	SOF-M3
`DGEMM`	Matrix-matrix multiplication	$2n^3$	$2L-1$	$L-1$	$2L-2$	$2L-2$
`DTRMM`	Triangular matrix-matrix multiplication	n^3			$2L-2$	$2L-2$
`DGEQP3`	Pivoted QR	$4/3n^3$	$L-1$			
`DGEQRF`	QR decomposition for a $2n \times n$ matrix	$10/3n^3$		$L-1$	$L-1$	$L-1$
`DGEQRF_M`	Modified `DGEQRF` to form T inside	$4n^3$				$L-1$
`DORGQR`	Form Q-factor after `DGEQRF`	$28/3n^3$		$L-1$		
`DLARFT`	Form T matrix after `DGEQRF`	n^3			$L-1$	

Algorithm 7 SOF step 2)-a) – distributed version 2

```
1) Perfom the QR decomposition using modified PDGEQRF
   to generate additional $T$ matrix
2) Compute $(Q_{12}^{(\ell)})^T = -V_d T^T V_u^T$, and
   $(Q_{22}^{(\ell)})^T = I - V_d T^T V_d^T$ using DGEMM and DTRMM
```

A. Stability and accuracy

The first experiment compares the correctness and numerical stability of the direct method, stratification method (Algorithm 1) and SOF-M1. The direct method for Green's function calculation first forms the product of B_is and then inverts the shifted product. The computed Green's functions by three methods are denoted G_{D}, G_{L} and G_{S} respectively. The stability and accuracy of SOF-M2 and SOF-M3 are essentially the same as SOF-M1.

Figure VI.3 shows the relative difference of three methods for computing the Green's function with a 4×4 ($n = 16$) periodic square lattice. The local repulsion parameter $U = 2$ and the number of time slices L varies from $L = 5$ to $L = 100$. The solid line is the difference of the direct method and stratification method: $\|G_{\text{D}} - G_{\text{L}}\|/\|G_{\text{L}}\|$. The crosses are for the stratification method and the SOF algorithm: $\|G_{\text{L}} - G_{\text{S}}\|/\|G_{\text{L}}\|$. The circled-line is for the direct method and the SOF algorithm. $\|G_{\text{L}} - G_{\text{S}}\|/\|G_{\text{L}}\|$. As can be seen, the difference of the direct method to the other two methods grows exponentially as L increases. On the other hand, the results of the stratification and SOF methods are consistent to machine precision even for large L.

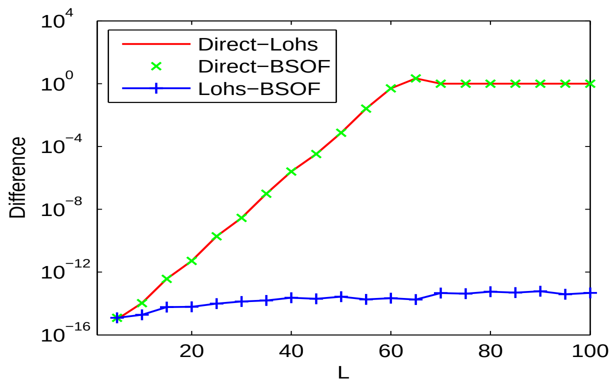


Fig. VI.3. Results for $U = 2$ and $L = 5, 10, \cdots, 100$.

The experiment result tells two things. First, the experiment shows the results computed by the stratification method and SOF-M1 method are essentially numerically identical. Second, the direct method is instable, although it is also an HPMO-avoiding algorithm. Without the stabilizing procedures, the produced result has no significant digit at all.

B. Performance on multicore

Let us compare the performance of the stratification method and the SOF methods an Intel Core i7 920 2.66GHz processor with Intel's `ifort` Fortran compiler, MKL BLAS and LAPACK libraries. The tested problem is the Green's function for periodic square lattices 32×32 ($n = 1024$), 48×48 ($n = 2304$), and 64×64 ($n = 4096$). The number of time slice is $L = 96$.

Table VI.2 displays the CPU execution time for the stratification method and three SOF implementations on multicore. The corresponding Gflop/s are listed in Table VI.3. From the results, one can see that on one core, the stratification method is the fastest one. However, when using more cores, the SOF algorithm shows its superiority. For all three cases, SOF-M2 and SOF-M3 are faster than the stratification method on quad-core. Figure VI.4 displays the relative execution time of SOF-M1, SOF-M2 and SOF-M3 to the stratification method for $n = 2304, L = 96$. The relative execution time is calculated as

$$T_i^{\text{rel}} = \frac{T_i - T_L}{T_L}, \qquad \text{(VI.8)}$$

where T_i is the execution time of SOF-Mi and T_L is the execution time of the stratification method. It can be seen that when 4 cores are used, SOF-M2 is more than 20% faster than the stratification method, and SOF-M3 is about 16% faster.

Although SOF-M1 is the slowest among those methods, it has better speedup than the stratification method. In other word, one can expect better scalability on parallel environments, which eventually can outperform the stratification method when the number of cores is increased.

C. Performance on distributed architecture

In this experiment, we compare the performance of the stratification method and the SOF implementations SOF-H1 and SOF-H2 on a Cray XT4 (named Franklin) with the number of processors 4×4, 8×8, 16×16, and 32×32. The platform description of Franklin is in section II.

TABLE VI.2
THE CPU EXECUTION TIME OF FOUR IMPLEMENTATIONS

problem size	method	1 core	2 cores	4 cores
32×32 (1024)	Stratification	95.93	65.56	51.23
	SOF-M1	195.96	110.91	69.66
	SOF-M2	139.68	74.34	43.94
	SOF-M3	139.88	76.73	46.32
48×48 (2304)	Stratification	1108.04	726.26	537.36
	SOF-M1	2004.81	1072.40	619.84
	SOF-M2	1464.60	764.11	426.90
	SOF-M3	1493.49	791.34	448.60
64×64 (4096)	Stratification	6452.46	3740.33	2633.87
	SOF-M1	10864.11	5698.11	3096.94
	SOF-M2	8161.95	4277.24	2348.55
	SOF-M3	8357.09	4406.83	2412.09

TABLE VI.3
PERFORMANCE IN GLOPS

problem size	method	1 core	2 cores	4 cores
32×32 (1024)	stratification	5.76	8.42	10.78
	SOF-M1	7.65	13.51	21.52
	SOF-M2	7.59	14.25	24.12
	SOF-M3	7.36	13.41	22.21
48×48 (2304)	stratification	5.68	8.66	11.70
	SOF-M1	8.51	15.92	27.54
	SOF-M2	8.24	15.79	28.27
	SOF-M3	7.83	14.78	26.08
64×64 (4096)	stratification	5.48	9.45	13.41
	SOF-M1	8.83	16.83	30.96
	SOF-M2	8.31	15.85	28.87
	SOF-M3	7.86	14.90	27.23

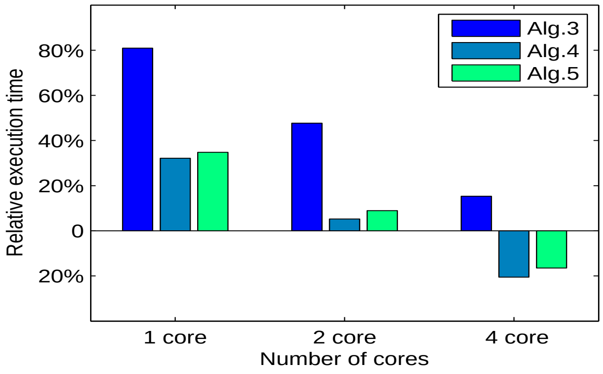


Fig. VI.4. Relative execution time of SOF-M1, SOF-M2 and SOF-M3 to stratification method for $n = 2304, L = 96$ on Intel Core i7 920.

TABLE VI.4
CPU EXECUTION TIME OF STRATIFICATION METHOD AND SOF-H1 AND SOF-H2 ON FRANKLIN.

Number of Processors	16 (4x4)	64 (8x8)	256 (16x16)	1024 (32x32)
stratification	354.36	127.13	41.12	53.23
SOF-H1	249.29	79.36	29.21	21.29
SOF-H2	158.35	57.75	28.55	26.79

The Green's function has the size 4096×4096 with $L = 12$. Table VI.4 shows the execution time of three methods.

The advantage of the SOF algorithm is clear on distributed architectures. One can see even with more floating point operations, SOF-H1 runs much faster than the stratification method, and scales better with more processors. The stratification method becomes slower when the processor number increases from 16×16 to 32×32.

The GFlops of the stratification method and two SOF implementations on Franklin are shown in Table VI.5 and Figure VI.5. One can see the performance of SOF-H1 and SOF-H2 are much higher than that of the stratification, which explains the faster running time of the SOF algorithm even with much larger operation counts. Theoretically, SOF-H2 uses less operations and should outperform SOF-H2. However, SOF-H2 was programmed without any performance tuning, which might be the reason that its performance is lower than that of SOF-H1.

VII. DISCUSSION AND FUTURE WORK

Numerical algorithm runtimes are increasingly dominated by the cost of memory access (communication), which can exceed the costs of floating point operations by orders of magnitude. Although this trend has been around for a while since the development of the high-level BLAS (i.e., BLAS 3) and its use in LAPACK and ScaLAPACK in 1980s and early 90s, it has continued to grow more important with the shift to an increasing number of cores and heterogeneous architectures. Significantly new and redesigned algorithms are needed to reduce and minimize communication costs [8]. In this paper, we demonstrate how to redesign a higher-level matrix algorithm by using those matrix operations that are easier to parallelize and their performance tuned software are widely available.

The performance of the SOF algorithm proposed in this paper depends on QR. The implementations we discussed here do not utilize any fine-grained and asynchronous techniques to explicitly form the partial Q-factor. A potential improving method is to design a multi-threaded subroutine that can fully utilize the power of multicore as those presented in [17], [7], [6], [15].

The idea of HPMO-avoidance discussed in this paper can be extended to other higher-level matrix algorithms that are critical to large scale scientific computing applications. For example, matrix polar decomposition is also another important matrix operation widely used in many applications, such as the subspace alignment in electronic structure calculation [10]. We plan to investigate a low-communication algorithm for computing the polar decomposition [16]. We also plan to revisit an inversion-free spectral divide and conquer algorithms for nonsymmetric eigenproblems [3].

ACKNOWLEDGMENT

The first author would like to acknowledge National Science Council of Taiwan for the support under the grant NSC98-2218-E-007-006-MY3, and National Center for High-performance Computing for using the computing facilities. This work is also supported in part by the Office of Science of the U.S. Department of Energy under SciDAC (Scientific

TABLE VI.5

GFLOPS/S OF THE STRATIFICATION METHOD AND TWO IMPLEMENTATIONS ON FRANKLIN.

Number of Processors	16 (4x4)	64 (8x8)	256 (16x16)	1024 (32x32)
stratification	12.64	35.23	108.92	84.14
SOF-H1	61.20	192.26	522.34	716.65
SOF-H2	70.68	193.80	392.02	417.77

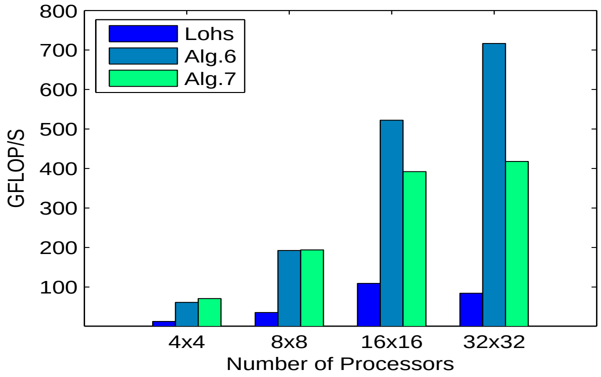


Fig. VI.5. Performance on Cray XT4 (NERSC Franklin)

Discovery through Advanced Computing) grant DE-FC-02-06ER25793. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] Marc Baboulin, Jack J. Dongarra, and Stanimire Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. Technical Report 200, LAPACK Working Note, May 2008.

[2] Z. Bai, W. Chen, R. Scalettar, and I. Yamazaki. Numerical methods for quantum monte carlo simulations of the Hubbard model. In T. Y. Hou *et al*, editor, *Multi-Scale Phenomena in Complex Fluids*, pages 1–110. Higher Education Press, 2009.

[3] Z. Bai, J. Demmel, and M. Gu. An inverse free parallel spectral divide and conquer algorithm for nonsymmetric eigenproblems. *Numer. Math.*, 76:279–308, 1997.

[4] Z. Bai, C-R. Lee, R.-C. Li, and S. Xu. Stable solutions of linear systems involving long chain of matrix multiplications. *Linear Algebra and its Applications*, 2010. doi:10.1016/j.laa.2010.06.023.

[5] R. Blankenbecler, D. J. Scalapino, and R. L. Sugar. Monte carlo calculations of coupled boson-fermion systems. I. *Phys. Rev. D 24,*, 24:2278–2286, 1981.

[6] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report 191, LAPACK Working Note, September 2007.

[7] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, September 2008.

[8] James W. Demmel, Laura Grigori, Mark F. Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report 204, LAPACK Working Note, August 2008.

[9] S. L. Graham, M. Snir, and C. A. Patterson, editors. *Getting Up To Speed: The Future Of Supercomputing*. National Academies Press, Washington, D.C., 2005.

[10] F. Gygi. Architecture of Qbox: A scalable first-principles molecular dynamics code. *IBM Journal of Research and Development*, 52(1-2):137–144, 2008.

[11] Nicholas J. Hiham. *Functions of Matrices: Theory and Computation*. SIAM, 2008.

[12] E.Y. Loh Jr. and J.E. Gubernatis. Stable numerical simulations of models of interacting electrons in condensed matter physics. In W. Hanke and Yu.V. Kopaev, editors, *Electronic Phase Transitions*, pages 177–235. Elsvier Sicence Publishers, 1992.

[13] E.Y. Loh Jr., J.E. Gubernatis, R.T. Scalettar, R.L. Sugar, and S.R. White. Stable matrix-multiplication algorithms for the low-temperature simulations of fermions. In D.K. Campbell D. Baeriswyl and, editor, *Interacting Electronics in Reduced Dimensions*. Plenum, 1989.

[14] C-R. Lee, I-H. Chung, and Z. Bai. Parallelization of determinant quantum monte carlo simulations for strongly correlated electron systems. In *Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*. IEEE, 2010.

[15] Hatem Ltaief, Jakub Kurzak, and Jack Dongarra. Parallel block Hessenberg reduction using algorithms-by-tiles for multicore architectures revisited. Technical Report 208, LAPACK Working Note, August 2008.

[16] Y. Nakatsukasa, Z. Bai, and F. Gygi. Optimizing Halley's iteration for computing the matrix polar decomposition. *SIAM J. of Matrix Anal. Appl.*, 31(5):2700–2720, 2010.

[17] G. Quintana-Orti, E. S. Quintana-Orti, R. A. van de Geijn, E. Chan, and F. G. van Zee. Programming algorithms-by-blocks for matrix computations on multithreaded architectures. *ACM Transactions on Mathematical Software*, 36(3), July 2009.

[18] G.W. Stewart. On graded QR decompositions of products of matrices. *Electronic Transactions in Numerical Analysis*, 3:39–49, 1995.

[19] G.W. Stewart. *Matrix Algorithms: Volume I Basic Decompositions*. SIAM, 1998.

[20] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of 2008 ACM/IEEE Conference on Supercompiting (SC'08)*. IEEE, 2008.