PERFORMANCE OPTIMIZATIONS FOR QUANTUM CHEMISTRY CALCULATIONS

A Thesis
Presented to
The Academic Faculty

By

Hua Huang

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
College of Computing

Georgia Institute of Technology

May 2019

PERFORMANCE OPTIMIZATIONS FOR QUANTUM CHEMISTRY CALCULATIONS

Approved by:

Dr. Edmond Chow, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. David Sherrill
School of Chemistry and Biochemistry, School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Phanish Suryanarayana
School of Civil and Environmental Engineering
Georgia Institute of Technology

Date Approved: April 17, 2019

This thesis is dedicated to my parents.

ACKNOWLEDGEMENTS

I would like to sincerely thank my parents, Deqiang and Yongqiu. Without your love, trust, and support, it is impossible for me to step forward and explore the world.

I would like to sincerely thank my advisor, Professor Edmond Chow, for his guidance and support. His belief in me provided encouragement throughout the entire process of my study and research. The wisdom and insights I learned from his words made much of this work possible. I'm very happy and excited that I can continue to be his PhD student in the next several years.

I would like to sincerely thank Professor David Sherrill and Professor Phanish Suryanarayana for their valuable suggestions on my research and serving on my thesis committee.

I gratefully acknowledge funding from an Intel Parallel Computing Center grant and the National Science Foundation grant ACI-1147843.

I would like to sincerely thank Benjamin Prichard, Jeff R. Hammond, Min Si, and Xing Liu for their assistance and helpful discussion.

I would like to sincerely thank Dr. Weicai Ye, my undergraduate advisor at Sun Yat-sen University. Thank you for changing my life by guiding me to the world of high-performance computing and pushing me to have my graduate study abroad.

I would like to sincerely thank Dr. Yunfei Du, chief engineer of National Supercomputer Center in Guangzhou. Thank you for sharing your insight into high-performance computing with me and giving career suggestions to me.

I would like to sincerely thank all of my friends around me and on the Internet. Thank you for sharing your time, happiness, and ideas with me. You make my life colorful.

I would also like to thank Mr. Anno Hideaki for delaying the production of movie *Shin Evangerion Gekijoban* : || for 8 years. The absence of this movie saves me hundreds of hours from watching the movie and discussing it on the Internet. Please try to keep your promise and release the movie in 2020.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Quantum chemistry is a mature area of computational science with many methods and codes developed that are used across chemistry, biochemistry, and materials science. Optimizing computational kernels in quantum chemistry calculations is usually challenging due to the high complexity of the algorithms and also the high complexity of modern computer hardware. This thesis focuses on optimizing the performance of three important computational kernels in quantum chemistry calculations.

We first optimize electron repulsion integral (ERI) calculations for Gaussian basis sets. A batching scheme for ERI calculations is designed that better utilizes vector processing units in a processor to calculate multiple ERIs simutaneously. With the optimized ERI calculations, the tensor contraction in Fock matrix construction can become the performance bottleneck. We design a thread-safe algorithm along with specific optimizations to improve the performance of shared-memory Fock matrix construction. For distributed-memory Fock matrix construction, we design a new portable partitioned global address space (PGAS) framework called GTMatrix. GTMatrix has better communication performance compared to the Global Arrays library which is a commonly used PGAS framework in quantum chemistry programs. Finally, we optimize density matrix purification, which is a method of constructing the density matrix directly from the Fock matrix. We present the new idea of *overlapping communications with communications* to accelerate matrix-matrix multiplications in density matrix purification.

We implement the optimizations in the GTFock library. GTFock is a high-performance Fock matrix construction library with a Hartree-Fock self-consistent field (SCF) demo program. Test results show that optimized GTFock is up to three times faster when performing an SCF iteration compared to the unoptimized version.

CHAPTER 1

INTRODUCTION

In quantum chemistry calculations, the Hartree-Fock (HF) method [1] is a fundamental and widely used method for approximately solving the electronic Schrödinger equation. In HF calculations, the construction of the Fock matrices (Coulomb and exchange matrices) and the density matrix consume more than 99% of the runtime. Fock matrices also arise in Density Functional Theory (DFT) [2], another highly accurate and popular method for electronic structure calculations. The construction of Fock matrices uses ERI calculation and tensor contraction kernels. The construction of density matrix requires eigenvalue computation. In this thesis, we focus on optimizing the performance of these three kernels.

Among these three kernels, ERI calculations are the most irregular and challenging to optimize when using the Guassian basis sets. An ERI describes the Coulombic interaction between two electrons in terms of four basis functions

$$(MN|PQ) = \int \phi_M(\vec{x}_1)\phi_N(\vec{x}_1)\frac{1}{r_{12}}\phi_P(\vec{x}_2)\phi_Q(\vec{x}_2)d\vec{x}_1 d\vec{x}_2 \tag{1.1}$$

where the $(MN|PQ)$ is notation denoting a specific ERI and r_{12} is the distance between the two electrons at $\vec{x}_1$ and $\vec{x}_2$. In particular, the ERI algorithms are recursive and loops have small trip counts, making efficient vectorization very difficult.

Previous attempts to improve the vector performance of ERI libraries took the approach of restructuring loops to make them vectorizable, annotating code to help the compiler auto-vectorizer, as well as hand-coding with intrinsics [3, 4]. However, the resulting vectorization efficiency was still poor, because the recursions and short loops of the ERI algorithms could not be addressed without a drastic code transformation or rewrite.

Recently, a library for ERI calculations, called Simint [5], was developed with the goal

obtaining high vector efficiency. Simint is based on the idea of simultaneously computing the components of an ERI, called *primitive integrals*, that have the same code path. Benchmarks showed superior performance of the Simint library over existing ERI libraries and this performance improvement could be directly attributed to improved vector performance.

However, ERIs may only have a single or small number of primitive integrals, which would give Simint little or no advantage over other ERI libraries in this case. Our solution here is to batch together the computation of multiple ERIs, simultaneously computing primitive integrals for these multiple ERIs, as long as the computation of these primitive integrals share the same code path. With enough primitive integrals, the vector loop computing the primitive integrals can become much more efficient.

Once ERIs are computed, they are used to construct the Fock matrix. Blocks of the Fock matrix F is constructed by contracting 4D ERI tensors with the density matrix

$$F_{MN} = H_{MN}^{core} + \sum_{PQ} D_{PQ}\{2(MN|PQ) - (MP|NQ)\}, \tag{1.2}$$

where H^{core} is a precomputed, fixed matrix, D is the density matrix, and $(MN|PQ)$ is a shell quartet of ERIs. In the past, the performance of Fock matrix accumulation has not been critical since the ERI calculations are dominant. With more efficient ERI calculation, the runtime of Fock matrix accumulation may exceed the runtime of ERI calculations. Thus, it is essential to optimize Fock matrix accumulation.

For shared-memory programs, the performance of Fock matrix accumulation has not been critical in the past since the ERI calculations are dominant. We found that with more efficient ERI calculation and larger numbers of threads, the runtime of shared-memory parallel Fock matrix accumulation may exceed the runtime of ERI calculations, especially when using basis sets that has small numbers of basis functions per atom [6]. We optimize the shared-memory parallel Fock matrix accumulation from three different aspects: (1) using a combination of multiple-level thread-local buffer and atomic operation to guarantee

thread-safety as well as reduce the usage of atomic operations, (2) packing the density matrix to increase data access locality, and (3) specializing kernels to reduce the vectorization overhead.

For distributed-memory programs, Fock matrix accumulation is usually performed using two steps: each process accumulates its ERI results into its local Fock matrix buffer, then the results in each process's local buffer are accumulated to obtain the final Fock matrix. The inter-process data exchange in the second step is usually handled by a partitioned global address space (PGAS) framework. We design a new PGAS framework called "GTMatrix" as a new option for distributed-memory Fock matrix construction. GTMatrix can also be used in other applications. GTMatrix is written in C and uses only MPI functions to provide fundamental functionality. However, test calculations show that GTMatrix can be up tp 45% faster compared to the Global Arrays [7] (GA) library.

After building the Fock matrix F, the density matrix D is constructed using the eigenvectors of F: let C_{occ} be a matrix formed by n_{occ} lowest energy eigenvectors of F, $D = XC_{occ}C_{occ}^{\mathrm{T}}X^{\mathrm{T}}$, where X is a precomputed, fixed basis transformation matrix. Traditionally, an eigendecomposition of F is performed to obtain C_{occ}, which is the performance bottleneck in many DFT codes and electronic structure methods. Instead of eigendecomposition, D can be computed directly from F by using the density matrix purification iteration [8]

$$D_{k+1} = 3D_k^2 - 2D_k^3, \tag{1.3}$$

where the initial approximation D_0 is an appropriately scaled and shifted version of F. This iteration has many practical variations, but all the variations involve matrix squaring and cubing, or forming even higher matrix powers. In linear scaling DFT, the density matrices are large and sparse [9]. In HF calculations, the density matrices have modest size in comparison and are best treated as being dense. In this case, eigendecomposition can be used to compute the density matrices, but eigendecomposition cannot scale as well as dense

matrix multiplication in purification methods when using thousands of processors [10].

The bottleneck of distributed-memory matrix multiplication is internode communication performance. To better exploit communication resources, we explore the idea of *overlapping communications with communications*. Here, communication operations are overlapped with other communication operations. This allows actual data transfer in one operation to be overlaped with synchronization or other overheads in another operation, thus making more effective use of the available network bandwidth. We discuss two techniques for overlapping communication operations in this thesis: nonblocking operation overlap and multiple MPI process per node (PPN) overlap. Our results will show that combining the two techniques appears to give the best performance results.

We implemented all these optimizations in GTFock [11, 10, 4], a recently developed high-performance library for Fock matrix construction. Test calculations were performed to show the effect of optimizations for each kernels and the overall speedup of Fock matrix construction and SCF iterations.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Vectorization of ERI Calculations

The solution to a quantum chemistry problem can be expressed in terms of basis functions ϕ_i, with $1 \leq i \leq n$ for n basis functions. "Gaussian" basis functions are most common, particularly for molecular systems, and are the target of this work, although other types of basis functions, such as plane waves and wavelets [12], are also used, particularly for materials systems.

Each Gaussian basis function is a linear combination (or "contraction") of known *primitive functions*, $\chi_{M\mu}$, i.e.,

$$\phi_M = \sum_{\mu}^{K_M} c_{M\mu} \chi_{M\mu}$$

(similarly for the other functions ϕ_N, ϕ_P, and ϕ_Q). Thus, the integral 1.1, which can be called a *contracted integral* for clarity, is the result of a four-fold sum,

$$(MN|PQ) = \sum_{\mu}^{K_M} \sum_{\nu}^{K_N} \sum_{\lambda}^{K_P} \sum_{\sigma}^{K_Q} c_{M\mu} c_{N\nu} c_{P\lambda} c_{Q\sigma} [\chi_{M\mu} \chi_{N\nu} | \chi_{P\lambda} \chi_{Q\sigma}]$$

where $[\chi_{M\mu} \chi_{N\nu} | \chi_{P\lambda} \chi_{Q\sigma}]$ in square brackets denotes a *primitive integral*. It is these primitive integrals involving recursive calculations that are difficult to vectorize. In the above example, the contracted integral is the sum of $K_M K_N K_P K_Q$ primitive integrals. Nominally, n^4 ERIs must be computed, a potentially very large number. Neglect of small ERIs, called *screening*, and exploiting symmetries such as $(MN|PQ) = (NM|PQ) = (PQ|NM)$ are necessary in practical implementations.

A *basis set* is a specification of the basis functions to use for different atomic species in a molecule. Some basis sets are *highly contracted*, meaning its basis functions are sums

of many primitive functions (up to a dozen or more), whereas others are *lightly contracted*, meaning its basis functions are mostly represented by a single primitive function.

A basis function is characterized by its *angular momentum* (AM), which can take on values $0, 1, 2, 3, 4, \ldots$, denoted as $s, p, d, f, g, \ldots$, respectively. The primitive functions comprising a basis function have the same AM as the basis function. Since an integral is associated with four basis functions, an integral is also associated with four AM numbers, e.g., $(ss|pd)$, where the bracket notation has been overloaded, and denotes the AM *class* of an integral.

Vectorization of ERI calculations has been considered since the era of pipelined vector supercomputers. On the CRAY-1, Saunders and Guest [13] proposed vectorizing primitive integral calculations for low AM cases, where the shell quartet structure, which complicates vectorization, could be ignored without too high a cost. For the high AM cases, it was suggested to vectorize the contractions. On the Alliant FX-8, Gill, Head-Gordon, and Pople [14] also proposed vectorizing primitive integrals. Here, contractions could be performed early or late in the algorithm depending on AM class. In the mid 1990s, Wolinski et al. [15] proposed batching contracted integrals in the TEXAS code to improve performance in the same vein as using BLAS constructs.

The above programs are legacy Fortran codes using common blocks. The TEXAS integral module, however, was recently converted to be used in a multithreaded environment [16] and is still the default integral library in NWChem [17]. Despite its design, however, Shan et al. [3] found vectorization performance to be very poor: "Via substantial programming effort, we obtained a vectorized version running approximately 25% faster compared to non-vectorization mode on the MIC and BG/Q platforms."

Libint2 [18], one of the integral libraries used in GAMESS [19], has experimental capability to vectorize across contracted integrals, but does not appear to be used this way. Vectorizing across contracted integrals poses several challenges, including the implementation of primitive integral screening, much larger working set size than vectorizing across

primitive integrals, and needing to not only batch integrals with the same AM class but also the same contraction pattern, K_M, K_N, K_P, K_Q.

In the last decade, there has also been extensive interest in computing ERIs on GPUs [20] [21] [22] [23] [24] [25]. Here, research addressed the use of single precision, the host-device bottleneck, and whether to vectorize primitive or contracted integrals. Some of the codes developed are limited to low AM functions as limited device memory makes simultaneous computation of large numbers of high AM integrals very challenging. These works also appeared to ignore practical issues such as integral screening and exploitation of symmetry, which would introduce thread divergence.

Ramdas and co-authors [26, 27, 28] discuss the advantages of batching ERIs of the same AM class, such as instruction and data cache exploitation, and propose batching algorithms that can be implemented on FPGAs, but there are few details on actual implementations.

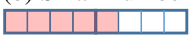
The Simint library vectorizes the computation of primitive integrals of the same class. For the computation of the specific ERI $(MN|PQ)$, the number of primitive integrals is $K_M K_N K_P K_Q$. Three cases are illustrated in Figure 2.1. For lightly contracted basis sets (a), the number of primitive integrals may be as small as 1, and thus no vectorization is possible. Even for moderately contracted basis sets (b), SIMD utilization may be poor because a large proportion of the integrals is calculated outside the SIMD loop. For highly contracted basis sets (c), SIMD utilization is good. Thus, for good performance for all basis sets, particularly lightly contracted basis sets, it is necessary to batch together *contracted integrals* of the same class and compute their primitive integrals in the same SIMD loop.

2.2 Parallel Fock Matrix Construction Algorithms

Practically most quantum chemistry software implement parallel Fock matrix construction subroutines. Some quantum chemistry software implement Fock matrix construction with distributed-memory computation to obtain better performance, for example, NWChem [17], GAMESS [19], ACESIII [29], and MPQC [30]. Some quantum chemistry packages like

(a) Single primitive integral: poor SIMD utilization

(b) Small number of primitives: generally poor SIMD utilization



(c) Large number of primitives: good SIMD utilization

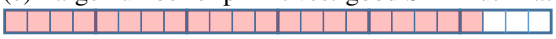


Figure 2.1: Different numbers of primitive integrals calculated using SIMD vectorization using the example of 4 doubles per SIMD word. (Primitive integrals occupying lanes of a SIMD word are shown with red shading.) For high SIMD utilization, a large number of primitive integrals of the same AM class are necessary.

PSI4 [31] and pySCF [32] do not have distributed-memory implementation, but they are easier to use and to extend with Python programming interfaces. In this section, we discuss two major challanges in parallel Fock matrix construction and their solutions.

2.2.1 Task Partitioning and Scheduling of ERI Calculations

The first challange of parallel Fock matrix construction is partitioning shell quartets that survive screening and uniqueness ("valid" shell quartets) to all processes and/or threads such that the workload on each process or thread is approximately balanced. Usually, shared-memory multithreaded programs solve this problem easily with a shared-memory dynamic task scheduler, for example, the OpenMP dynamic loop scheduler. For distributed-memory parallel programs, this problem is the key for parallel scalability.

For distributed-memory Fock matrix construction, there are two ways to distribute the valid shell quartets to different processes. The first approach is using a dynamic scheduling algorithm to schedule "workload units" to each process, where each workload unit is a set of shell quartets. This approach is easy to balance the load and it is used by NWChem and GAMESS. In NWChem, a minimal work unit is defined to be valid shell quartets that all shells belong to 5 atoms. In GAMESS, a work unit is defined as a 2D slice $(M,N|:,:)$ of the 4D shell quartet tensor for the pure MPI version, or a 3D slice $(M,:|:,:)$ of the 4D shell quartet tensor for the MPI+OpenMP version [33]. This approach has two major problems: (1) the centralized dynamic scheduler is very likely to be a performance bottleneck when

the program is run on a large number of cores, and (2) since the workload unit scheduling is completely dynamic, processes cannot prefetch all the blocks of D used in a workload unit. The second approach is to statically partition the valid shell quartets among processes. The advantage of this approach is that the necessary communication is known before calculation starts. Therefore, the partitions can be chosen to reduce the communication volume and each process can prefetch all D blocks needed by this process. However, load balance is difficult to achieve in this approach.

To be run on a large number of nodes efficiently on supercomputers, GTFock uses a hybrid approach for ERI calculation task partitioning and scheduling. In GTFock, a static partitioning with approximately balanced workload is used to reduce communication, a dynamic work stealing stage is used to polish the load balance. Specifically, a node steals work from another node when it finishes its allocated work. A basic work unit used by the work stealing scheduler is called a *task*, a 2D slice of the 4D shell quartet tensor:

$$\begin{aligned}&(\mathscr{M}_i,:|\mathscr{P}_k,:) \equiv \{(MN|PQ), \text{s.t.}\\ &M \in \mathscr{M}_i, P \in \mathscr{P}_k, \text{for all } N,Q\},\\ &i \in \{1,\ldots,p_r\}, j \in \{1,\ldots,p_c\}.\end{aligned}$$

The static partitioning of GTFock seeks to find a set of $\mathscr{M}_i$ and $\mathscr{P}_k$ s.t. the number of non-screened shell quartets in each task is better balanced. To achieve this, GTFock first calculates the *significant set*, which is based on the shell quartet screening, of each shell:

$$\sigma(P,Q) = \max_{i\in P, j\in Q}(ij|ij),\ m^* = \max_{P,Q}\ \sigma(P,Q),$$
$$\Phi(M) = \{N \text{ s.t. } \sigma(M,N) \geq \tau^2/m^*\},$$

and $\eta(M)$ is the size of $\Phi(M)$. $\eta(M)\eta(P)$ is an upper bound and a good estimation of the number of non-screened shell quartets in slice $(M,:|P,:)$. For p^2 processes and k^2 tasks per process, the shells can be divided into kp subsets, with the i-th subset denoted as $\mathscr{G}_i$.

GTFock choose the subsets such that the sum of the $\eta(M)$ for each subset is approximately the same,

$$\sum_{M \in \mathscr{G}_i} \eta(M) \approx (\sum \eta(M))/p.$$

The partitions of task can be indexed by (i, j) with $i, j \in [1, kp]$. Then slice $(M, : |P, :)$ is assigned to task (i, j) if $M \in \mathscr{G}_i, P \in \mathscr{G}_j$. Thus, the number of shell quartets survived screening is approximated balanced in each task.

GTFock uses a hierarchical stealing scheme for efficient dynamic work scheduling. The process in GTFock are arranged in a logical $p_r \times p_c$ process grid. A global array W is used to indicate which process group still have works. All elements in W is set to 1 at the beginning. When a process need to steal a task, it will first find a process group that still have works. Then, a victim process is selected randomly in the chosen process group and half of the victim's remaining tasks are stolen. Compared to directly find a victim process and steal its work, this hierarchical stealing scheme largely reduce the number of failed steals and the communication cost of task stealing.

2.2.2 Parallel Fock Matrix Construction Using ERI Results

The second problem of parallel Fock matrix construction is accumulating ERI results in parallel to build the Fock matrix and handling the race condition in the parallel accumulation. The data access patterns in constructing the Fock matrix is highly irregular due to the intrinsic structure of molecules and the shell quartet screening. Therefore, it is hard to know the update time of any given block in the Fock matrix in advance and totally remove the race condition when multiple processes or threads are updating the Fock matrix at the same time.

For shared-memory programs, two approaches can be used for Fock matrix construction: (1) each thread accumulates its ERI results into a thread-private Fock matrix buffer

and sums the results in all thread-private buffers in the last step, and (2) each thread accumulates its ERI results to a shared Fock matrix using atomic operations. The PSI4 package [31] implemented the first approach and uses atomic operations for summing thread-private results. The GAMESS package implemented both approaches in its MPI+OpenMP version. Performance tests show that the private Fock buffer approach is faster than the shared Fock matrix approach in GAMESS due to less thread contention; as the penalty, the private Fock buffer approach has larger memory consumption and poorer multi-node scalability compared to the shared Fock matrix approach [33].

For distributed-memory programs, Fock matrix accumulation is usually performed using two steps: each process accumulates its ERI results into its local Fock matrix buffer, then the results in each process's local buffer are accumulated to obtain the final Fock matrix. The inter-process data exchange in the second step is usually handled by a PGAS framework, for example, the Global Arrays library.

2.3 Parallel Matrix Multiplication Algorithms

Parallel matrix multiplication can be categorized into 2D, 3D, and 2.5D algorithms. In 2D algorithms, a 2D partitioning of the matrix is used and processes are organized in a 2D mesh. The Scalable Universal Matrix Multiplication Algorithm (SUMMA) [34] is the most widely used 2D algorithm.

To reduce the communication cost, 3D algorithms [35, 36] use a 3D partitioning of work and organize the processes in a 3D mesh. These algorithms use more memory than 2D algorithms; each input matrix is replicated across one dimension of the process mesh. Compared to 2D algorithms, the communication cost of 3D algorithms is reduced from $O(N^2/P^{1/2})$ to $O(N^2/P^{2/3})$, where N is the matrix dimension and P is the number of processes. As a trade-off, the memory requirement of 3D algorithms is raised from $O(N^2/P)$ to $O(N^2/P^{2/3})$.

2.5D algorithms [37] bridge the gap between 2D and 3D algorithms, allowing the user

to choose how much memory to use to reduce communication. A parameter c is introduced in 2.5D algorithms to control the number of replicated copies of the input matrices, up to the limit corresponding to 3D algorithms.

The algorithms described so far are designed for general dense matrices. For general sparse matrix multiplication, SUMMA has been extended in SpSUMMA [38, 39], using doubly compressed sparse column (DCSC) storage format and sparse generalized matrix multiplication (SpGEMM) for local matrix multiplication. Matrix multiplication for block-rank-sparse matrices in quantum chemistry have also been developed using a task-based approach [40].

CHAPTER 3
ACCELERATING ERI CALCULATIONS WITH VECTORIZED AND BATCHED INTEGRALS

3.1 Simint Low-level Optimizations for Vectorization

Simint is a vectorized implementation of the Obara-Saika (OS) algorithm [41, 42, 43, 14, 44, 45] for computing ERIs. In the OS algorithm, an integral with angular momentum (AM) class $(ij|kl)$ can be computed using recurrence relations involving *auxiliary* integrals with lower AM class, i.e., involving functions with lower AM. To give a conceptual but not precise description below, we denote an auxiliary integral of class $(ij|kl)$ as $\Theta^{(N)}_{ijkl}$ where the index N equals 0 for the desired target integral.

To compute $\Theta^{(0)}_{ijkl}$, multi-dimensional recurrence relations are used, where the base of the recursions are $\Theta^{(N)}_{0000}$ for any value of N, which are called Boys functions of order N, and which can be computed directly. The recurrence relations can be organized into vertical recurrence relations (VRR) and horizontal recurrence relations (HRR) and can be used as follows.

To compute $\Theta^{(0)}_{ijkl}$, one begins by computing $\Theta^{(m)}_{i+j,0,0,0}$ via a recurrence relation known as a bra-side (i.e., left-hand side) VRR, and then computing $\Theta^{(m)}_{i+j,0,k+l,0}$ via a ket-side (i.e., right-hand side) VRR, followed by computing $\Theta^{(m)}_{i,j,k+l,0}$ via a bra-side HRR, and finally by computing $\Theta^{(m)}_{i,j,k,l}$ by a ket-side HRR. For reference, these recurrence relations are:
Bra-side VRR:

$$\Theta^{(N)}_{i+1,0,0,0} = X_{PA}\Theta^{(N)}_{i,0,0,0} - \frac{\alpha}{p}X_{PQ}\Theta^{(N+1)}_{i,0,0,0} + \frac{i}{2p}\left(\Theta^{(N)}_{i-1,0,0,0} - \frac{\alpha}{p}\Theta^{(N+1)}_{i-1,0,0,0}\right)$$

Ket-side VRR:

$$\Theta^{(N)}_{i,0,k+1,0} = X_{QC}\Theta^{(N)}_{i,0,k,0} - \frac{\alpha}{q}X_{PQ}\Theta^{(N+1)}_{i,0,k,0} + \frac{k}{2q}\left(\Theta^{(N)}_{i,0,k-1,0} - \frac{\alpha}{q}\Theta^{(N+1)}_{i,0,k-1,0}\right) + \frac{i}{2(p+q)}\Theta^{(N+1)}_{i-1,0,k,0}$$

Bra-side HRR:

$$\Theta^{(N)}_{i,j+1,k,l} = \Theta^{(N)}_{i+1,j,k,l} + X_{AB}\Theta^{(N)}_{i,j,k,l}$$

Ket-side HRR:

$$\Theta^{(N)}_{i,j,k,l+1} = \Theta^{(N)}_{i,j,k+1,l} + X_{CD}\Theta^{(N)}_{i,j,k,l}.$$

In the above, X_{PA}, etc., are parameters that depend on the atomic coordinates of the molecular system, and p, q, α, are parameters of the basis functions. The index m above denotes an appropriate range of indices for auxiliary integrals that must be computed.

Simint applies the VRRs to SIMD words rather than regular double-precision words, i.e., to 8 doubles at a time in the case of AVX-512. One could also apply the HRRs to SIMD words, however, since the HRRs do not involve parameters that depend on the basis functions, the primitive integrals can be contracted at this point, before the HRRs, thus saving computation and storage. Thus the HRRs are not perfectly vectorized, but some auto-vectorization of the bra-side HRR is possible. Consequently, the vector efficiency of Simint depends on the AM class, with classes involving more VRRs than HRRs being more efficient.

3.1.1 Optimizing General Functions for High AM Integrals

Simint code is generated, with separate functions for computing integrals of each AM class. For low AM classes, the recursive operations are expanded explicitly and inlined. For high AM classes, in order to reduce code size, general functions are used to implement the VRRs and HRRs involving high AM functions. The following two optimizations are simple, although they could be easily missed, and give some performance improvement.

Listing 3.1: General bra-side HRR function

```
void HRR_J_f_d(const int ncart, double *hAB,
    double *fdXX, double *gpXX, double *fpXX)
{
    for (int iket = 0; iket < ncart; ++iket)
    {
        fdXX[0*ncart+iket]=gpXX[0*ncart+iket]+hAB[0]*fpXX[0*ncart+iket];
        fdXX[1*ncart+iket]=gpXX[3*ncart+iket]+hAB[1]*fpXX[0*ncart+iket];
        fdXX[2*ncart+iket]=gpXX[6*ncart+iket]+hAB[2]*fpXX[0*ncart+iket];
        fdXX[3*ncart+iket]=gpXX[4*ncart+iket]+hAB[1]*fpXX[1*ncart+iket];
        // more calculations (no simple pattern)
    }
}
```

General Bra-side HRR Functions

As an example, the Simint function *HRR_J_f_d()* computes auxiliary integrals of AM class $(fd|**)$ from those of classes $(fp|**)$ and $(gp|**)$ using a bra-side HRR. These functions can be auto-vectorized by the compiler, but timings show that some AM classes with only VRRs and bra-side HRRs have poor vectorization speedup, and that the general bra-side HRR functions consume most of the time. These general bra-side HRR functions accept a parameter, *ncart*, to specify the number of basis functions in a shell on the ket side, and this parameter determines how data is accessed (see Listing 3.1). If this parameter value is known at compile time, auto-vectorization could be improved.

The parameter *ncart* only takes on a small set of values. Therefore, we have created specialized versions of *HRR_J_f_d()* and other functions for specific values of *ncart*. These specialized versions are called from a wrapper function. The general version is kept for large values of *ncart*, which are rare. As a result, the compiler can compute all the offsets of output auxiliary integrals and know the length of the loop at compile time, which reduces the cost of the loop and leads to better vectorization.

Listing 3.2: contract_all() implementation for AVX-512

```
inline void contract_all(int ncart, __m512d const *src, double *dest)
{
    for (int np = 0; np < ncart; np++)
        dest[np] += _mm512_reduce_add_pd(src[np]);
}
```

General VRR Functions

In Simint, the function *ostei_general_vrr_K()* implements a ket-side VRR. Profiling found that this function is a hotspot when the basis set has many high AM shells. This function is implemented in a general way to open algorithmic options: it can compute $(i, j|k+1, l)$ for any given $i, j, k, l \geq 0$. However, the ket-side VRR in the OS algorithm assumes targets of the form $(i, 0|k+1, 0)$. By specializing this function for this case, branches that determine which recursions to take can be reduced from 16 to 4 and can also be moved from the innermost loop to an outer loop. The original general function is retained for use with different algorithmic options.

3.1.2 Optimizing Contractions for AVX-512

The contraction operation sums primitive auxiliary integrals to form a contracted auxiliary integral. In Simint, the *contract_all()* function performs, for a set of SIMD words, a horizontal reduction of double-precision words in a SIMD word, as shown in Listing 3.2. Each SIMD word corresponds to a different auxiliary integral, not the same integral. The *contract_all()* function was measured to be a computational hotspot.

In the AVX-512 case shown in Listing 3.2, *contract_all()*, naturally uses the intrinsic function *_mm512_reduce_add_pd()* to sum the components of a SIMD vector. However, analyzing the assembly code for *contract_all()* reveals two problems: (1) *_mm512_reduce_add_pd()* does not have a corresponding CPU instruction, but is emulated by 8 instructions; (2) when $ncart > 10$, the compiler does not unroll the loop.

Considering that the AVX-512 instruction set has new shuffle instructions and that the

Listing 3.3: Optimized contract_all() implementation for AVX-512

```
1  inline void contract_all(int ncart, __m512d const *src, double *dest)
2  {
3      int ntrans   = ncart / 8;
4      int np_start = ntrans * 8;
5      double tmp[64];
6      __m512d dst[8];
7      // Transpose-Add part
8      for (int it = 0; it < ntrans; it++)
9      {
10         double *src_ptr = (double*)src + it * 64;
11         for (int i = 0; i < 8; i++)
12         {
13             for (int j = 0; j < 8; j++)
14                 tmp[i * 8 + j] = src_ptr[j * 8 + i];
15             dst[i] = _mm512_load_pd(tmp + i * 8);
16         }
17         __m512d res = _mm512_loadu_pd(dest + it * 8);
18         for (int i = 0; i < 8; i++)
19             res = _mm512_add_pd(res, dst[i]);
20         _mm512_storeu_pd(dest + it * 8, res);
21     }
22     // Remainder part
23     for (int np = np_start; np < ncart; np++)
24         dest[np] += _mm512_reduce_add_pd(src[np]);
25 }
```

Intel compilers can utilize these instructions efficiently [46] [47], we use a novel approach to accelerate the *contract_all()* function. For a 8×8 block of double-precision words (8 SIMD words), we first transpose the block, then perform vectorized add for the transposed vertical vectors. Listing 3.3 is the new, optimized implementation. Lines 11 - 14 transpose a 8×8 block, and line 15 effectively hints to the compiler that the transposed data will be used immediately and should be kept in registers. As a quick comparison, for a 8×8 block, the optimized implementation needs 47 CPU instructions, while the original approach needs $16 \times 8 = 128$ CPU instructions. Since *ncart* is not always a multiple of 8, we use the original approach for the remainder part.

We also make a special optimization for AM class $(ss|ss)$, since *contract_all()* for $(ss|ss)$ is a large portion of the runtime when using basis sets with few high AM functions. The subroutine for computing $(ss|ss)$ calls *contract_all()* with $ncart = 1$, so the remainder part

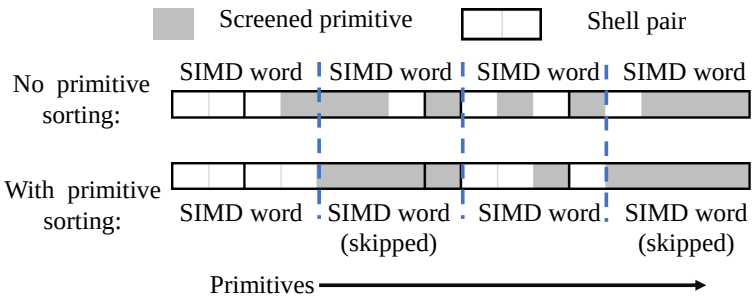


Figure 3.1: Sorting primitive integrals for primitive screening. Heavy lines delineate primitive integrals corresponding to a ket-side shell pair. An example is shown for a sequence of 5 ket-side shell pairs with 2, 5, 1, 3, and 5 primitive integrals, respectively. Without sorting, no SIMD words can be skipped. With sorting, two of the four SIMD words can be skipped. All four primitives in a SIMD word are screened (shown in gray) when a SIMD word can be skipped.

in Listing 3.3 would be used. Instead, to use the new approach, we gather 8 SIMD words corresponding to the same contracted integral, then store the results separately. It should be noted that this gather-contract-store approach can also be used for other AM classes, but it would make the code much more complicated, and the cost of extra operations in these cases may cancel out the saved time.

3.1.3 Sorting for Primitive Screening

Primitive screening is the concept of neglecting the computation of primitive integrals when they are known to be small. Since the computation of the primitive integrals is vectorized, primitive screening creates "holes" (screened primitives) in the SIMD words, which are not handled efficiently. Simint only neglects the computation of primitive integrals if all primitives in a SIMD word are screened. To improve vectorization efficiency, we sort all primitives in a shell pair in descending order according to an upper bound based on the Cauchy-Schwarz inequality. If a primitive involving a shell pair is screened, all primitives behind it will also be screened. As a result, all neglected primitives will be placed together, which increases the probability that all primitives in a SIMD word are screened. Figure 3.1 shows the mechanism of sorting for primitive screening in Simint.

3.2 Batching ERI Calculations to Improve Vectorization

In distributed memory quantum chemistry codes, the set of shell quartets to be computed is partitioned statically or dynamically among the compute nodes. Within a node, each thread computes and consumes the integrals for a shell quartet, one shell quartet at a time. As described in Chapter II, SIMD utilization in Simint for computing ERIs may be poor when shell quartets are computed one at a time. In this Section, we describe a batching procedure for shell quartets of the same AM class that is executed on each node. Each thread will thus compute the integrals for multipleshell quartets as one unit of work and thus improve SIMD utilization, particularly for lightly contracted basis sets.

The batching procedure has several requirements:

1. Only unique shell quartets are computed, i.e., $(MN|PQ)$ is symmetric with 7 other shell quartets, for example $(NM|QP)$, and only one of these should be computed.

2. Shell quartets containing integrals that all are small in magnitude are neglected and the shell quartet is not computed; this is called *shell quartet screening* and is different from and used together with primitive integral screening.

3. Batching cannot precompute a list of all shell quartets with the same AM class, as the large number of shell quartets makes this infeasible.

4. Batching for a given AM class should be handled by a single thread; this maximizes the number of shell quartets that can be batched together.

3.2.1 ERI Batching Scheme

Our solution is to use a dynamic procedure where queues for each AM class are maintained. Shell quartets are added to the appropriate queue depending on their AM class, and a full queue constitutes a batch of shell quartets with integrals to be computed.

Algorithm 1 shows the procedure. For simplicity, we assume an appropriate partitioning of the shell quartets among the nodes and that the indices M, N, P, Q are only over the indices for quartets that are assigned to a node. The M,N loop (line 2) is parallelized with OpenMP multithreading. Each thread maintains its private shell quartet queues, pushing all valid shell quartets it encounters to a queue according to the AM of the P and Q shells (lines 4 - 6). A batch of shell quartets will be submitted to Simint when a queue is full, then the ERI results are consumed, and this queue is reset (lines 7 - 10). When a thread has looped over all P,Q pairs for a given M,N pair, it submits all its non-empty queues to Simint and resets these queues (lines 13 - 16). We note that the procedure is thread-aware but lock-free.

Algorithm 1 Batched ERI computation

```
 1: Each thread initializes its private queues
 2: for shell pairs M,N in parallel do
 3:    for shell pairs P, Q do
 4:       if (MN|PQ) is unique and not screened then
 5:          Compute bra-side shell pair id: id = (P,Q)
 6:          Push (MN|PQ) to queue q[id]
 7:          if queue q[id] is full then
 8:             Compute shell quartets in queue q[id]
 9:             Reset queue q[id]
10:          end if
11:       end if
12:    end for
13:    for a non-empty queue q[i] do
14:       Compute shell quartets in queue q[i]
15:       Reset queue q[i]
16:    end for
17: end for
```

The shell quartet queues in Algorithm 1 should be long enough to provide good SIMD efficiency, but very long queues are not necessary. In the experimental tests (see Table 7.6 later in this thesis), we found a queue length of 16 to give good performance.

3.2.2 ERI Library Issues for Batched Computation

To compute the integrals for a batch of shell quartets, Algorithm 1 calls Simint with the handle of shell pair (M,N), and a set of handles to shell pairs $\{(P_i,Q_i)\}$ such that all shells P_i have the same AM and all shells Q_i have the same AM. This should be considered the basic unit of work for a vectorized integral library.

The computation of the ERIs in a shell quartet $(MN|PQ)$ uses some quantities that only depend on *shell pairs* (M,N) and (P,Q). These quantities, called *shell pair data*, can be precomputed and stored because they are used for multiple shell quartets. If we store shell pair data, we must gather the appropriate data into a continuous buffer for batched ERI computation, since the vectorized computation needs the data in this format. Alternatively, shell pair data can be regenerated every time they are needed.

Let p_1 and p_2 be the number of primitive functions for two shells in a shell pair. The store-reuse approach needs to copy $12 \times p_1 \times p_2$ double-precision words for this shell pair, where 12 is the number of arrays in its shell pair data. Regeneration of the shell pair data only needs to read the coordinates and coefficients data for the two shells, which is $6 + 2 \times (p_1 + p_2)$ double-precision words. With primitive screening without primitive sorting, the regeneration approach could be a little faster than the store-reuse approach in some cases.

To see which approach is more efficient, we tested both these approaches with different workloads. Experiments showed that storing and reusing all shell pair data is much faster in most cases. If we disable sorting for primitive screening, recomputing shell pair data is a little bit faster in a few cases. Therefore, we use the store-reuse approach.

Finally, we note a caveat when using batching with a dynamic distribution of shell quartets to the compute nodes. When dynamic distribution is used, a task containing some number of shell quartets is assigned to a node that is free. When ERI computations are batched, one must make sure that the tasks contain enough shell quartets such that large enough batches of the same AM class can be formed.

CHAPTER 4

EFFICIENT SHARED-MEMORY FOCK MATRIX ACCUMULATION

In practice, the Fock matrix is not constructed one block at a time as Equation 1.2 suggests. Due to the high cost of ERI calculation, a unique shell quartet $(MN|PQ)$ is computed once, which is equivalent to computing $(NM|PQ)$ and 6 other symmetric shell quartets, and then contributes to F_{MN}, F_{MP}, F_{NP}, F_{MQ}, F_{NQ}, F_{PQ} are accumulated into F. These six blocks are all the combinations of choosing pairs of the 4 shell indicies, M, N, P, Q. Since F is symmetric, F_{NM}, F_{PM}, etc. do not need to be accumulated.

Algorithm 2 shows the basic Fock matrix accumulation procedure after a shell quartet $(MN|PQ)$ is computed. In the algorithm, *ERI* denotes the 4-D array storing the results of ERIs in the shell quartet, with dimensions $dimM \times dimN \times dimP \times dimQ$. A four-fold loop iterates over each element in the shell quartet and computes contributions to F. We note that if the maximun AM of the shell quartets in a given problem is 4, which is not uncommon, $dimX \in [1,3,6,10,15], X = M,N,P,Q$. Discussion in the rest parts of this section is based on this algorithm.

4.1 Thread-safe Fock Matrix Accumulation

Since multiple threads are computing shell quartets and accumulating them into a shared Fock matrix F, a thread-safe Fock matrix accumulation algorithm is needed. Assuming one copy of F for some set of threads, for thread safety, atomic operations are used to update F_{PQ}, F_{MQ}, and F_{NQ} (lines 9-11) in Algorithm 2, resulting in $3 \times dimM \times dimN \times dimP \times dimQ$ atomic operations. Updates to F_{MN}, F_{MP} and F_{NP} can be accumulated in registers, with atomic operations used to accumulate these register values outside the iQ

Algorithm 2 Fock matrix accumulation, given shell quartet $(MN|PQ)$ with dimensions $dimM \times dimN \times dimP \times dimQ$.

```
1: for iM = 0 to dimM-1 do
2:   for iN = 0 to dimN-1 do
3:     for iP = 0 to dimP-1 do
4:       for iQ = 0 to dimQ-1 do
5:         I = ERI(iM, iN, iP, iQ)
6:         Update F_MN(iM, iN) with D_PQ(iP, iQ),  I
7:         Update F_MP(iM, iP) with D_NQ(iN, iQ),  I
8:         Update F_NP(iN, iP) with D_MQ(iM, iQ), I
9:         Update F_PQ(iP, iQ) with D_MN(iM, iN), I
10:        Update F_MQ(iM, iQ) with D_NP(iN, iP),  I
11:        Update F_NQ(iN, iQ) with D_MP(iM, iP), I
12:      end for
13:    end for
14:  end for
15: end for
```

loop to update F. The total number of atomic operations needed here is

$$AO_1 = 3 \times dimM \times dimN \times dimP \times dimQ + 2 \times dimM \times dimN \times dimP + dimM \times dimN.$$

Our paramount consideration is reducing the usage of atomic operations, which we find to limit the performance of this approach. Thus, an obvious alternative is to split Algorithm 2 into six four-fold loops such that each four-fold loop only updates one block of the Fock matrix. The order of the loops can be exchanged so that atomic operations can be placed in the second nested loop. Algorithm 3 is an example for F_{MQ}. The calculation and update of other Fock matrix blocks are similar. After splitting, the total number of atomic operations becomes the sum of the sizes of six updated blocks

$$AO_2 = dimM \times (dimN + dimP + dimQ) + dimN \times (dimP + dimQ) + dimP \times dimQ.$$

Algorithm 3 Updating F_{MQ} with a separate four-fold loop

```
1: for iM = 0 to dimM-1 do
2:    for iQ = 0 to dimQ-1 do
3:       register f_MQ = 0
4:       for iN = 0 to dimN-1 do
5:          for iP = 0 to dimP-1 do
6:             I = ERI(iM, iN, iP, iQ)
7:             Update f_MQ with D_NP(iN, iP) and I
8:          end for
9:       end for
10:      atomic_add(F_MQ(iM, iQ), f_MQ)
11:   end for
12: end for
```

As the price of reducing the number of atomic operations, the performance of this new approach is redundant memory access: it needs to read the entire *ERI* array six times and has discontinuous memory access to *ERI* when updating F_{MQ}, F_{NQ} and F_{PQ}. We want to find a way that uses only AO_2 atomic operations while reading *ERI* continuously only once.

We observe that AO_2 is relatively small in most cases. If the maximum AM of the shells in a given problem is 4, which is not uncommon, then the maximum size of any dimension of the *ERI* array is 15, and thus $AO_2 \leq 15^2 \times 6 = 1350$ doubles. Each thread can use a thread-local buffer to accumulate the updates of six Fock submatrices and add them to the shared F at the end. Therefore, we do not need to split the four-fold loop in Algorithm 2 and we can avoid reading *ERI* multiple times. The buffer for each thread is small enough ($<$ 11 KB for maximum AM = 4) to fit in the L1 data cache of most processors.

When the ERIs are computed in batched fashion, another optimization allows us to eliminate about half of the atomic operations. In the batched ERI algorithm, line 3 of Algorithm 1 actually contains two loops: the outer loop iterates over shell indices P and the inner loop iterates over shell indices Q. As a result, all shell quartets in a batch have the same M and N shells and are likely to have only a small number of different P shells.

Suppose that all shell quartets in a batch have the same M, N and P shells. In this case, all shell quartets in the batch update the same F_{MN}, F_{MP}, F_{NP}. Therefore, the accumulation

of F_{MN}, F_{MP}, F_{NP} can be performed in the thread-local buffers without atomics until the end of the batch. Atomics are only used at the end of the batch to accumulate these thread-local buffers into the shared F. As before, the accumulation of F_{PQ}, F_{MQ}, F_{NQ} needs atomics for each shell quartet processed. Thus the number of atomic operations can be reduced to

$$AO_3 = dimQ \times (dimM + dimN + dimP)$$

per shell quartet other than the last quartet in a batch. We present this optimized Fock matrix accumulation procedure in Algorithm 4.

Stepping further from Algorithm 4, we can further reduce the usage of atomic operations and speed up Fock matrix accumulation by using multi-level thread-private buffer. We call this new approach "MLBufAcc" for short. Recall the definition of a compute task in GTFock (see Section 2.2.1). Let U_{PQ} be the total number of unique P index and unique Q index in a compute task. $U_{PQ} \leq 2 \times nshell$, where *nshell* is the total number of shells in the chemical system. More importantly, U_{PQ} is much smaller than the total number of ket-side shell pair indices $|PQ)$ in a compute task in most cases. Therefore, for a fixed bra-side shell pair indices $(MN|$, the total number of unique blocks update by F_{MN}, F_{MP}, F_{NP}, F_{MQ}, and F_{NQ} is usually much smaller than the total number of ket-side shell pair indices in a compute task. MLBufAcc uses two band-shape thread-local buffer FM and FN to hold the accumulations to F_{MN}, F_{MP}, F_{NP}, F_{MQ}, and F_{NQ}. Once all shell quartets with the same $(MN|$ indices are processed, FM and FN are accumulated to the shared F using atomic operations and reset to 0. For F_{PQ}, MLBufAcc still uses a thread-private block buffer for it and accumulates the block buffer to the shared F once a shell quartet. Algorithm 5 shows the MLBufAcc algorithm. In MLBufAcc, FM and FN are of size $maxDim \times nbf$, where *maxDim* is the maximum size of any dimension of the *ERI* array and nbf is the total number of basis functions. Thus, the size of FM and FN are of $O(nbf)$ level. This is much smaller than using private F copies of $O(nbf^2)$ size.

Algorithm 4 Fock matrix accumulation using thread-local buffer, for shell quartet $(MN|PQ)$

if MN has changed since last call **then**
  Initialize thread-local buffer f_{MN} to 0
end if
if MP and NP have changed since last call **then**
  Initialize thread-local buffers f_{MP}, f_{NP} to 0
end if
Initialize thread-local buffers f_{MQ}, f_{NQ}, f_{PQ} to 0
for iM = 0 to $dimM$-1 **do**
  for iN = 0 to $dimN$-1 **do**
    for iP = 0 to $dimP$-1 **do**
      for iQ = 0 to $dimQ$-1 **do**
        $I = ERI(iM, iN, iP, iQ)$
        Update $f_{MN}(iM, iN)$ with $D_{PQ}(iP, iQ)$, I
        Update $f_{MP}(iM, iP)$ with $D_{NQ}(iN, iQ)$, I
        Update $f_{NP}(iN, iP)$ with $D_{MQ}(iM, iQ)$, I
        Update $f_{PQ}(iP, iQ)$ with $D_{MN}(iM, iN)$, I
        Update $f_{MQ}(iM, iQ)$ with $D_{NP}(iN, iP)$, I
        Update $f_{NQ}(iN, iQ)$ with $D_{MP}(iM, iP)$, I
      end for
    end for
  end for
end for
Update F_{MQ}, F_{NQ}, F_{PQ} with f_{MQ}, f_{NQ}, f_{PQ}
if last MP and NP **then**
  Update F_{MP}, F_{NP} with f_{MP}, f_{NP}
end if
if last MN **then**
  Update F_{MN} with f_{MN}
end if

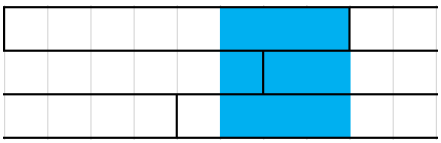
Algorithm 5 Fock matrix accumulation for shell quartet $(MN|PQ)$ using multi-level buffer

```
1: Initialize thread-ptivate buffer f_PQ to 0
2: for iM = 0 to dimM-1 do
3:    for iN = 0 to dimN-1 do
4:       for iP = 0 to dimP-1 do
5:          for iQ = 0 to dimQ-1 do
6:             I = ERI(iM, iN, iP, iQ)
7:             Update FM_N(iM, iN) with D_PQ(iP, iQ),  I
8:             Update FM_P(iM, iP) with D_NQ(iN, iQ),  I
9:             Update FN_P(iN, iP) with D_MQ(iM, iQ), I
10:            Update FM_Q(iM, iQ) with D_NP(iN, iP),  I
11:            Update FN_Q(iN, iQ) with D_MP(iM, iP), I
12:            Update f_PQ(iP, iQ)  with D_MN(iM, iN), I
13:         end for
14:      end for
15:   end for
16: end for
17: Update F_PQ with f_PQ using atomic operations
18: if this PQ is the last pair in ket-side shell pairs then
19:    Update F with FM and FN using atomic operations
20:    Reset FM and FN to 0
21: end if
```

4.2 Increasing Memory Access Locality in Fock Matrix Accumulation

The Fock matrix accumulation procedure needs to access six small blocks in the Fock matrix and the density matrix. In modern processors, data is transferred between memory and cache in blocks of fixed size called "cache line". The typical cache line size for CPU is 64 bytes. Assuming the data type of matrix element is *double*, fetching a row of a 3-by-3 block (24 bytes) from memory requires transferring one or two cache lines (64 or 128 bytes) to the cache. Figure 4.1 shows a possible situation of memory transfer when accessing this 3-by-3 block. We can see that the memory locality of accessing these small blocks from a large matrix is very poor.

To increase the memory locality of accessing the Fock and density matrix, we use a block storage scheme for the Fock matrix and density matrix. Both the Fock matrix and density matrix can be decomposed into non-overlapping blocks and the position of each



Black boundary: 64B cache line (8 double word)
Blue block: a 3 × 3 submatrix, need 4 cache lines

Figure 4.1: Possible memory transfer when accessing a 3-by-3 block from a large matrix.

block can be calculated from the shell pair indices associated with this block. Therefore, the Fock matrix and density matrix can be packed such that the elements of each block is stored consecutively and the block for shell pair $(M,N+1)$ is stored next to the block for shell pair (M,N). Density matrix is packed before each Fock matrix construction, and the Fock matrix is unpacked at the end of Fock matrix construction.

4.3 Reduce Vectorization Overhead in Fock Matrix Accumulation

After reducing the usage of atomic operations, the performance of Fock matrix accumulation is still bounded by two factors: (1) the flop-per-byte ratio of Fock matrix accumulation is low, and (2) $dimQ$ is usually veary small (basis sets usually have more low AM shells than high AM shells), which leads to a significant vectorization overhead in Fock matrix accumulation. If the maximun AM of the shells in a given problem is 4, $dimQ = 1$ and $dimQ = 3$ are more common that $dimQ \in [6,10,15]$. To reduce the vectorization overhead in Fock matrix accumulation, we create five specialized kernels of Fock matrix accumulation for specific value of $dimQ$, and call the general kernel when $dimQ \geq 21$. In these specialized kernels, we require the compiler to unroll the iQ loop to avoid vectorization overhead. In the general kernel, we require the compiler to vectorize the iQ loop.

CHAPTER 5

PORTABLE PGAS FRAMEWORK FOR DISTRIBUTED-MEMORY FOCK MATRIX BUILD

For distributed-memory Fock matrix construction, a partitioned global address space (PGAS) framework is usually used to handle inter-process data exchange and dynamic task scheduling. The intrinsic structure of molecules and the shell quartet screening make the data access patterns in Fock matrix construction highly irregular. Using a PGAS framework, a program can index and locate the data it needs in a straightforward way, which lowers the difficulty of development and improves maintainability.

Previously, GTFock used the Global Arrays [7] (GA) library as the PGAS framework. GA uses ARMCI [48] and ComEx [49] as its underlying communication infrastructure and provides operations of different abstraction levels:

1. Primitive communication operations including get, put, and accumulate,
2. Basic matrix operations including transposing, scaling and symmetrizing a matrix,
3. High-level operations including matrix-matrix multiplication and eigendecomposition.

GA is used in applications in many subject areas, including computational fluid dynamics (NWGrid [50] and STOMP [51]), bioinformatics (ScalaBLAST [52]), and quantum chemistry (NWChem [17], GAMESS [33] and Molpro [53]).

Considering that GTFock only uses a small subset of operations provided in GA, we built a lightweight PGAS library called "GTMatrix" as a new option for GTFock. GTMatrix provides only fundamental functionality: tiled storage and access of a global matrix, distributed task counter, matrix-matrix multiplication, and symmetrizing a matrix. GTMatrix is written in C and only uses MPI functions for communication. GTMatrix does not

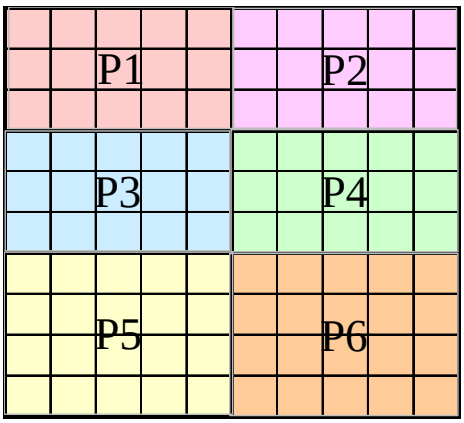


Figure 5.1: GTMatrix created in Listing 1

use third-party library and does not have hardware-specified optimization. Therefore, GTMatrix is lightweight and portable. It is designed for programs that need relatively simple functionality and provides a platform to study communication performance. For a full set of features, Global Arrays or UPC++ [54] are available.

5.1 C Interface of GTMatrix

GTMatrix provides a C interface with a syntax that is similar to the syntax of Global Arrays. Listing 5.1 shows the usage of GTMatrix. In Listing 5.1, lines 3 - 11 create a 10×10 global matrix *bm* with *double* data type. *bm* is a GTMatrix handle. The global matrix is partitioned into 3×2 blocks and all MPI processes in the MPI communicator *bm* store one block as Figure 5.1 shows. Users can control the row and column displacements of each block using *r_displs* and *c_displs* arrays.

GTMatrix provides two modes for accessing a block of a global matrix: the single blocking access mode and the batched nonblocking access mode. The parameters are the same for both modes. When a single blocking access function returns, the access is completed. When a batched nonblocking access function returns, the access request is stored and will be executed when a batch executing function is called. In Listing 5.1, line 23 uses the single blocking access mode to fetch a block $(rs0 : rs0 + rn0 - 1, cs0 : cs0 + cn0 - 1)$

Listing 5.1: Example Code of Using GTMatrix

```
// Create a 10 * 10 matrix with double datatype,
// distributed on a 3 row * 2 column process grid
GTMatrix_t gtm;
int nrow = 10, ncol = 10;
int n_rowblk = 3, n_colblk = 2;
int r_displs[4] = {0, 3, 6, 10};
int c_displs[3] = {0, 5, 10};
GTM_createGTMatrix(
    &gtm, mpi_comm, MPI_DOUBLE, sizeof(double),
    nrow, ncol, n_rowblk, n_colblk, &r_displs[0], &c_displs[0]
);

// Local matrix block is stored in gtm->mat_block in row-major
// style with leading dimension gtm->ld_local. Initialize the
// local matrix block according to your algorithm.

// Synchronize all MPI processes of a GTMatrix to make sure
// all MPI processes completely initialize their local block
GTM_Sync(gtm);

// Fetch a block [rs0:rs0+rn0-1, cs0:cs0+cn0-1] from global matrix to
// local row-major buffer buf0 with leading dimension ld0
GTM_getBlock(gtm, rs0, rn0, cs0, cn0, buf0, ld0);
// Synchronize all MPI processes of a GTMatrix to make sure
// all MPI processes get the initial data of the global matrix
GTM_Sync(gtm);

// ========== Local Computation ==========

// Accumulate multiple local blocks to global matrix
GTM_startBatchUpdate(gtm);
for (int i = 0; i < num_acc_blk; i++)
    GTM_addAccBlockRequest(gtm, rs[i], rn[i], cs[i], cn[i], buf[i], ld[i]);
GTM_execBatchUpdate(gtm);
GTM_stopBatchUpdate(gtm);

// ========== Save results ==========

// Destroy the GTMatrix structure and exit
GTM_destroyGTMatrix(gtm);
```

(MATLAB colon notation is used here). When *GTM_getBlock* returns, the data is fetched and stored in a local row-major matrix *buf*, and *ld* is the leading dimension of *buf*. Line 19 is used to synchronize the MPI processes in a GTMatrix to make sure each MPI process has initialized its local matrix block before any MPI process can reach line 23. Line 26 is used to prevent some MPI processes from updating the global matrix in lines 31 - 35 before other MPI processes' fetching the initial global matrix in line 23. After local computation, lines 31 - 35 use the batched access to accumulate multiple local blocks to the global matrix. *GTM_startBatchUpdate* declares that this MPI process is starting a batch nonblocking update and cannot use *GTM_addGetBlockRequest* before *GTM_stopBatchUpdate* is called. The accumulations in lines 32 - 33 are not completed until *GTM_execBatchUpdate* returns. List 5.2 lists the C interface of GTMatrix and the functionality of each function.

5.2 Design of GTMatrix

GTMatrix uses MPI one-sided communication functions and passive target synchronization for communication. For one-sided read, write, and update operations, GTMatrix uses *MPI_Get*, *MPI_Put*, and *MPI_Accumulate*, respectively. GTMatrix uses passive target synchronization instead of active target synchronization for two reasons: (1) the accesses are truly one-sided: access operations are totally handled by the source process (the process that posts the one-sided access), and (2) it allows an MPI application to have "the widest portability and performance" [55].

GTMatrix uses MPI derived data types (DDTs) to obtain better performance in accessing matrix blocks. MPI DDTs are used to tell the MPI library how to pack data from or unpack data to one or more basic data types. Using MPI DDTs allows the MPI library to handle the data packing automatically and avoid unnecessary data copying on some hardware [56]. GTMatrix uses MPI DDTs in two ways. For small matrix block access, GTMatrix predefines a set of MPI DDTs to reduce the overhead of creating and releasing MPI DDTs. For large matrix block access, a new MPI DDT is defined and used just-in-time

Listing 5.2: C interface of GTMatrix

```c
// ========== Constructor and destructor ==========
// Create and initialize a GTMatrix structure
void GTM_createGTMatrix(
    GTMatrix_t *gt_mat, MPI_Comm comm, MPI_Datatype datatype,
    int unit_size, int nrows, int ncols, int r_blocks, int c_blocks,
    int *r_displs, int *c_displs
);
// Destroy a GTMatrix structure
void GTM_destroyGTMatrix(GTMatrix_t gt_mat);

// Data access functions in GTMatrix has the same parameters,
// we define it as a marco here for convenience.
#define GTM_PARAM \
    GTMatrix_t gt_mat, int row_start, int row_num, \
    int col_start, int col_num, void *src_buf, int src_buf_ld

// ========== Fetching matrix block ==========
// Get a block
void GTM_getBlock(GTM_PARAM);
// Add a request to get a block
void GTM_addGetBlockRequest(GTM_PARAM);
// Start a batch get epoch and allow to submit update requests
void GTM_startBatchGet(GTMatrix_t gt_mat);
// Execute all get requests in the queues
void GTM_execBatchGet(GTMatrix_t gt_mat);
// Stop a batch get epoch
void GTM_stopBatchGet(GTMatrix_t gt_mat);

// ========== Updating matrix block ==========
// Put a block
void GTM_putBlock(GTM_PARAM);
// Add a request to put a block
void GTM_addPutBlockRequest(GTM_PARAM);
// Accumulate a block
void GTM_accumulateBlock(GTM_PARAM);
// Add a request to accumulate a block
void GTM_addAccumulateBlockRequest(GTM_PARAM);
// Start a batch update epoch and allow to submit update requests
void GTM_startBatchUpdate(GTMatrix_t gt_mat);
// Execute all update requests in the queues
void GTM_execBatchUpdate(GTMatrix_t gt_mat);
// Stop a batch update epoch
void GTM_stopBatchUpdate(GTMatrix_t gt_mat);

// ========== Helper functions ==========
// Symmetrize a matrix, i.e. (A+A^T)/2,
// now supports int and double data type
void GTM_symmetrizeGTMatrix(GTMatrix_t gt_mat);
// Synchronize all processes
void GTM_Sync(GTMatrix_t gt_mat);
```

and released after posting the access operation.

GTMatrix uses MPI shared memory to accelerate intra-node MPI process communication. When creating a global matrix, GTMatrix creates a shared memory MPI window and a global MPI window. For read operations, if target processes (the processes that the target data located in) are in the same node as the source process, GTMatrix uses direct memory copy in the shared memory MPI window instead of *MPI_Get*. For write and update operations, GTMatrix always uses *MPI_Put* and *MPI_Accumulate* in the global MPI window to guarantee the atomicity of these operations.

The batched access mode in GTMatrix is designed to accelerate: (1) large numbers of accesses to the same process, and (2) many-to-many communications. When a program submits an access request to GTMatrix in batched access mode, the access request is decomposed into multiple single-target access requests such that each of these new requests has only one target process. Then, each single-target access request is pushed to its target process's queue. Access requests are posted and completed when the program calls GTMatrix to perform the batched access. The source MPI process only need to synchronize with each target process once for finishing all access requests instead of synchronizing for each access request. A ring algorithm is used to accelerate many-to-many communications using GTMatrix. A source process P_s completes its access requests to target process P_t in this order: $t = s, s+1, \ldots, p-1, p, 1, 2, \ldots, s-2, s-1$.

The distributed task counter in GTMatrix uses *MPI_Fetch_and_op* to perform read-and-increment operations on remote processes and atomic fetch-and-add operations for fast local counter access and update. We encapsulate these low-level MPI and atomic operations into a distributed task counter for easier usage. The dynamic task scheduler in GTFock uses this functionality to perform task stealing.

5.3 Using GTMatrix Efficiently

Using GTMatrix in GTFock and other MPI applications is easy. Nevertheless, users should choose an access mode carefully to maximize the communication performance of a communication procedure in the application. In a communication procedure:

- If some processes need to access one or several large blocks of the global matrix, we suggest using the batched access mode in this procedure to utilize the ring algorithm for large-volume many-to-many communications.
- If all processes need to access one or several moderate or small blocks of the global matrix, we suggest using the single blocking access mode which has a smaller process overhead compared to the batched access mode in this case.
- If some processes need to access many(≥ 5) blocks of the global matrix, the batched access mode should be used to reduce the synchronization cost of these accesses as well as utilize the ring algorithm for many-to-many communications.

As an example, now we anaylze three major communication procedures in GTFock and choose the GTMatrix access modes for each of them:

- **GatherD**: each process gets a large block of the density matrix that will used in Fock matrix construction. We use batched access mode for this procedure.
- **AccFBuf**: each process accumulates its private Fock matrix buffer to its public Fock matrix buffer or another process's Fock matrix buffer after constrcuting its private Fock buffer with ERI results in an ERI computation task. Since each process only need to update data on one process at a time, we use single blocking access mode for this procedure.
- **ScatterF**: each process accumulates its public Fock matrix buffer to corresponding blocks of the Fock matrix. Since each process needs to update a large number of varying-size blocks, we use the batched access mode for this procedure.

CHAPTER 6
ACCELERATING DENSITY MATRIX PURIFICATION VIA OVERLAPPING COMMUNICATION WITH COMMUNICATIONS

In density matrix purification, the computation kernel is calculating the square and cube of a symmetric matrix D_k in parallel. We call this kernel "SymmSquareCube" for short. GTFock uses the 3D matrix multiplication algorithm, which is proved to be communication optimal [37] when having enough memory, for SymmSquareCube. Nevertheless, network communication is still the bottleneck of the 3D matrix multiplication algorithm. Thus, we explore the idea of *overlapping communications with communications* to better utilize network bandwidth and accelerate communication operations in SymmSquareCube.

6.1 Techniques for Overlapping Communications

6.1.1 Using Nonblocking MPI Operations to Pipeline and Overlap Communications

In the new "nonblocking overlap" technique for overlapping communication operations, data to be communicated is divided into multiple parts and communicated using separate MPI communicators, i.e., each MPI process uses multiple MPI communicators, with each communicator performing communication simultaneously with other communicators. The communications can also be pipelined, as we will show in our first example below and in our dense matrix computation.

The rationale for nonblocking overlap is to keep communication units busy and to try to fully utilize the available network bandwidth by overlapping network data transfer in one communication operation with processing stages that have little network data transfer in other communication operations.

We explain how this new technique works with the following example. Consider the

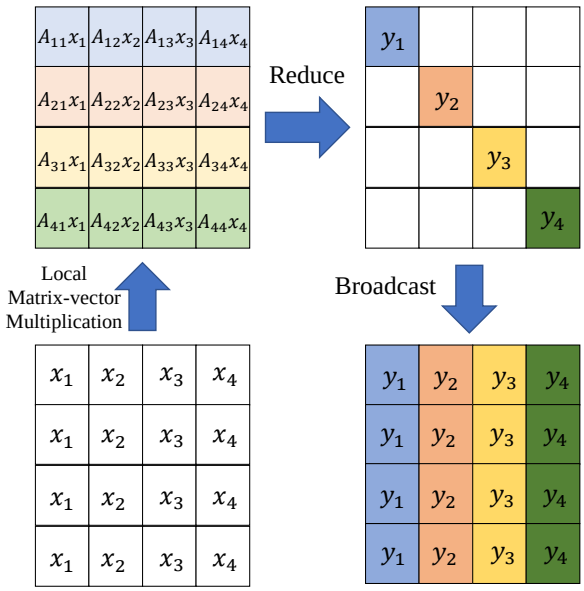


Figure 6.1: Communication operations in Algorithm 6 for a 4×4 process mesh. Colors denote different blocks of the vector y.

parallel matrix-vector multiplication $y = Ax$, where A is a $N \times N$ matrix and x and y are vectors. Matrix A is partitioned and distributed onto a $p \times p$ process mesh, vector x is partitioned into p blocks and processes $P_{:,i}$ have the i-th block of x (Matlab colon notation is used to specify mesh slices). After performing local matrix-vector multiplication, we need to reduce the local results to form the global result and then distribute y in the same way as x. Let *row_comm* denote a row communicator for processes $P_{i,:}$ and let *col_comm* denote a column communicator for processes $P_{:,i}$. Algorithm 6 is the algorithm for the parallel matrix-vector multiplication. Figure 6.1 illustrates the communications in the algorithm for a 4×4 process mesh.

Algorithm 6 Parallel matrix-vector multiplication

Input: A, x, *row_comm* and *col_comm* for all i
Output: y distributed as x

1: $P_{i,j}$ performs local matrix-vector multiplication: $y_i^{(j)} = A_{ij}x_j$ for all i, j
2: $P_{i,:}$ reduce sum $y_i^{(j)}$ to y_i on $P_{i,i}$ with *row_comm* for all i
3: $P_{i,i}$ broadcast y_i to $P_{:,i}$ with *col_comm* for all i

In Algorithm 6, the communicated data in lines 2-3 can be divided and the operations

Algorithm 7 Parallel matrix-vector multiplication with pipelined and overlapped communications

Input: A, x and $NDUP$ copies of both *row_comm* and *col_comm* for all i
Output: y distributed as x

1: $P_{i,j}$ performs local matrix-vector multiplication: $y_i^{(j)} = A_{ij}x_j$ for all i, j
2: Divide $y_i^{(j)}$ into $NDUP$ equal-size contiguous parts
3: **for** $c = 1$ to $NDUP$ **do**
4: $P_{i,:}$ posts the reduce sum of c-th part of y_i on $P_{i,i}$ with c-th *row_comm* using *MPI_Ireduce* for all i
5: **end for**
6: **for** $c = 1$ to $NDUP$ **do**
7: $P_{i,i}$ waits for completing the reduction of c-th part of y_i in line 4 using *MPI_Wait* for all i
8: $P_{i,i}$ posts the broadcasts of c-th part of y_i to $P_{:,i}$ with k-th *col_comm* using *MPI_Ibcast* for all i
9: $P_{i,j}$ posts the receive of c-th part of y_i broadcasted by $P_{j,j}$ with c-th *col_comm* using *MPI_Ibcast* for all $i \neq j$
10: **end for**
11: Wait for all outstanding *MPI_Ibcast* in lines 8 and 9 to finish

can be pipelined: $P_{i,i}$ can start broadcasting a segment of reduced y_i while still waiting for the reduction of the rest of y_i to be completed. Therefore, line 2 can be split and overlapped with line 3. Given $NDUP$ copies of *row_comm* and *col_comm*, overlapping the communications in lines 2 and 3 of Algorithm 6 with nonblocking operations gives Algorithm 7. Figure 6.2 shows the communications in Algorithm 7 for a 4×4 process mesh and $NDUP = 2$.

In Algorithm 7, line 4 posts nonblocking reductions for segments of y_i. Processes $P_{i,i}$ wait at line 7 for the completion of these reductions. Upon each completion, a segment of y_i is broadcast (lines 8-9) within the column communicator. Finally, all processes wait for the completion of the broadcasts.

For the same parallel program, there may be several ways of using the nonblocking overlap technique to optimize communication operations. Some principles for using this new technique efficiently are as follows:

- A parallel program may have several communication operations that can be put ad-

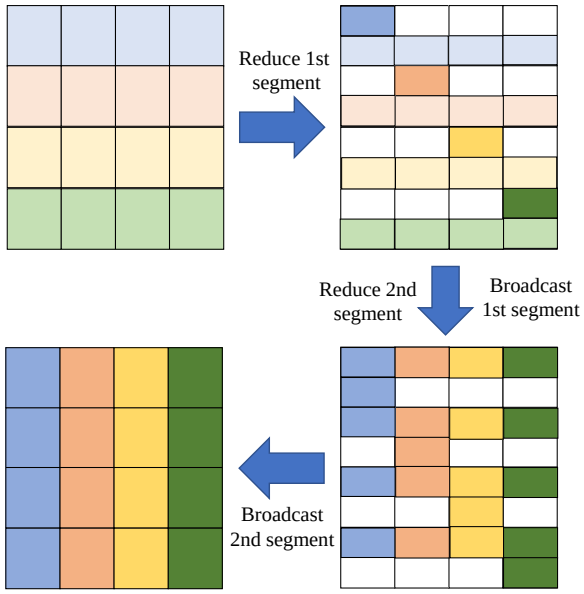


Figure 6.2: Overlapped and pipelined communication operations in Algorithm 7 for a 4×4 process mesh and $NDUP = 2$. Colors denote different blocks of vector y.

jacent to each other without changing the algorithm logic. One can split and overlap a single communication operation with itself, but having more communication operations gives us more opportunities to further overlap the communications.

- Collective operations should be overlapped. Collective operations have more execution stages and higher cost compared to point-to-point operations, which means that the potential performance gain can be larger if collective operations are overlapped.

- The data should be contiguous and the data layout should remain unchanged in the pipelined and overlapped communication operations. The extra cost of repacking data for the next operation may cancel out the benefit of pipelining and overlapping communication operations.

Choosing a proper value for $NDUP$ is also important. One can use different $NDUP$ values for different operations, if these operations are not overlapped with each other. The best $NDUP$ value could be different for different operations, and the best value should be chosen according to the size of the communicated data. When the message size is small, the communication time is dominated by network latency and the effective network bandwidth

is low. With a larger message size, the time consumed by data transfer becomes a larger portion of the communication time and the actual bandwidth is closer to the achievable bandwidth. Let the actual bandwidth BW_e be a function of message size n: $BW_e = f_{BW}(n)$. After applying nonblocking overlap, the message size is reduced by a factor of $1/NDUP$ and $NDUP$ operations are issued and pipelined. The actual inter-node bandwidth may not be as high as $NDUP$ times $f_{BW}(\frac{n}{NDUP})$. To further utilize the network bandwidth,

$$NDUP \cdot f_{BW}\left(\frac{n}{NDUP}\right) \geq f_{BW}(n)$$

is a necessary condition. An easier way to choose $NDUP$ is to make sure $n/NDUP$ is larger than or equal to a threshold value n_t, where $f_{BW}(n_t)$ is close to the achievable network bandwidth. For different machines, n_t may have different values, and usually 16 KB $\leq n_t \leq$ 1 MB.

If $n/NDUP \leq n_t$, using the nonblocking overlap technique is still possible and likely to accelerate communications, since some communication operations may utilize the network bandwidth while other communication operations are synchronizing or performing local processing. In this situation, using a very large $NDUP$ (such as 16) may give some speedup over using a small or moderate value (such as 4), but using a very large value of $NDUP$ would heavily consume system resources and have a large overhead.

6.1.2 Using Multiple PPN to Overlap Communications

The "multiple PPN overlap" technique is simply to run an application using multiple processes per node. The communication operations running on the separate processes are naturally overlapped and the communication resources may be better utilized. However, when increasing the number of processes per node, the following factors may increase and negatively affect performance, particularly for collectives:

- synchronization cost of blocking collective operations,

- number of steps in collective operations,
- inter-process communication and total communication volume.

Choosing the number of processes per node is a standard way to tune application performance. However, altering the number of processes per node changes multiple quantities simultaneously, largely (1) per-process quantities such as local problem size, data layout, number of threads, and memory access patterns, and (2) per-node quantities such as number of processes accessing the network interface. What may be optimal in one case may not be optimal for another. Thus, in order to make effective use of the multiple PPN overlap technique, we recommend combining it with the nonblocking overlap technique. The nonblocking overlap technique does not have the side-effects of changing the above quantities when the number of PPN is altered. Results for tests that combine the two techniques are shown in Section 7.5.

In application codes that are composed of different kernels, the optimal number of PPN for each kernel may be different. This is especially true given the complex interactions mentioned above when changing the number of PPN. The optimal number of PPN may also be different for computations (per-process effects of PPN) and communications (per-node effects of PPN) within one kernel. To gain finer control over the number of PPN at different stages of an application code, and for overlapping communication operations in the context of this thesis, we advocate a mechanism where many processes are launched per node and utilizing just the right number of these processes for each stage of the code. In this mechanism, in order to reduce or avoid explicit intra-node, inter-process data movement when the number PPN changes, the shared-memory features of MPI-3 could be used.

We implemented this mechanism for the density matrix purification kernel in our quantum chemistry code in order to choose the number of PPN for the purification kernel separately from the other kernels in the code. At the beginning of the purification kernel, processes that will be inactive call *MPI_Ibarrier*. Then these processes use *MPI_Test* and *usleep* functions to check for the wake-up signal (completion of the barrier) every 10 mil-

liseconds. Processes that are active perform the work of the purification kernel and then call *MPI_Ibarrier* when they are finished, in order to release the inactive processes and move collectively to the next kernel. Whether a process is active or inactive in a kernel depends on the number of PPN, which in turn is chosen to optimize the performance of that kernel.

6.2 Optimizing Matrix Squaring and Cubing

We use the name "SymmSquareCube" to denote the kernel for computing the square and cube of a symmetric matrix. Below, we describe a version of SymmSquareCube based on 3D matrix multiplication.

Four MPI communicators are used in SymmSquareCube: *global_comm* contains all p^3 MPI processes that participate in SymmSquareCube, *row_comm* contains processes $P_{:,j,k}$, *col_comm* contains processes $P_{i,:,k}$, and *grd_comm* contains processes $P_{i,j,:}$. The input matrix D is partitioned into $p \times p$ blocks and initially process $P_{i,j,1}$ has block $D_{i,j}$. The resulting D^2 and D^3 need to be partitioned and stored in the same way as D. Algorithm 8 is the original algorithm for SymmSquareCube released in the GTFock code. Algorithm 8 is slightly different from performing the standard 3D algorithm twice, in order to avoid unnecessary communication when D^2 and D^3 are both desired. In the first matrix multiplication, D acts as both matrix A and B in $C := A \times B$, and the broadcast direction of B is different from that of the standard 3D algorithm. In the second matrix multiplication, D again acts as A and D^2 acts as B, and only D^2 needs to be broadcast. The three broadcasts (lines 1, 2 and 7) and the two reductions (lines 4 and 9) are the most time consuming parts of Algorithm 8. The symmetry of D is only used in line 2.

6.2.1 Using the Nonblocking Overlap Technique

Algorithm 8 has three communication phases: lines 1-2, lines 4-7, and lines 9-10. Communications in each of these phases can be pipelined and overlapped. However, lines 5 and 6 are irregular point-to-point communications; the potential speedup of overlapping them

Algorithm 8 Original SymmSquareCube algorithm

Input: D, *row_comm*, *col_comm* and *grd_comm*
Output: D^2, D^3 distributed as D

1: $P_{i,j,1}$ broadcasts $D_{i,j}$ as $A_{i,j}$ to $P_{i,j,:}$ using *MPI_Bcast* with *grd_comm*
2: $P_{i,j,i}$ broadcasts $D_{i,j}$ as $B^{\mathrm{T}}_{j,i}$ to $P_{:,j,i}$ using *MPI_Bcast* with *row_comm*
3: Local matrix multiplication: $C_{i,j,k} := A_{i,j} \times B_{j,k}$
4: Reduce sum $C_{i,:,k}$ to $D^2_{i,k}$ on $P_{i,k,k}$ using *MPI_Reduce* in *col_comm*
5: $P_{i,k,k}$ sends $D^2_{i,k}$ to $P_{i,k,1}$ using *MPI_Send* and *MPI_Recv* in *grd_comm*
6: Transpose D^2 blocks s.t. $P_{k,j,k}$ has $D^2_{j,k}$ using *MPI_Send* and *MPI_Recv* in *global_comm*
7: $P_{k,j,k}$ broadcast $D^2_{j,k}$ as $B_{j,k}$ to $P_{:,j,k}$ using *MPI_Bcast* with *row_comm*
8: Local matrix multiplication: $C_{i,j,k} := A_{i,j} \times B_{j,k}$
9: Reduce sum $C_{i,:,k}$ to $D^3_{i,k}$ on $P_{i,k,k}$ using *MPI_Reduce* in *col_comm*
10: $P_{i,k,k}$ sends $D^3_{i,k}$ to $P_{i,k,1}$ using *MPI_Send* and *MPI_Recv* in *grd_comm*

Algorithm 9 Baseline SymmSquareCube algorithm

Input: D, *row_comm*, *col_comm* and *grd_comm*
Output: D^2, D^3 distributed as D

1: $P_{i,j,1}$ broadcasts $D_{i,j}$ as $A_{i,j}$ to $P_{i,j,:}$ using *MPI_Bcast* with *grd_comm*
2: $P_{i,j,i}$ broadcasts $D_{i,j}$ as $B^{\mathrm{T}}_{j,i}$ to $P_{:,j,i}$ using *MPI_Bcast* with *row_comm*
3: Local matrix multiplication: $C_{i,j,k} := A_{i,j} \times B_{j,k}$
4: Reduce sum $C_{i,:,k}$ to $D^2_{i,k}$ on $P_{i,i,k}$ using *MPI_Reduce* in *col_comm*
5: $P_{j,j,k}$ broadcast $D^2_{j,k}$ as $B_{j,k}$ to $P_{:,j,k}$ using *MPI_Bcast* with *row_comm*
6: Local matrix multiplication: $C_{i,j,k} := A_{i,j} \times B_{j,k}$
7: Reduce sum $C_{i,:,k}$ to $D^3_{i,k}$ on $P_{i,k,k}$ using *MPI_Reduce* in *col_comm*
8: $P_{i,i,k}$ sends $D^2_{i,k}$ to $P_{i,k,1}$ using *MPI_Send* and *MPI_Recv* in *global_comm*
9: $P_{i,k,k}$ sends $D^3_{i,k}$ to $P_{i,k,1}$ using *MPI_Send* and *MPI_Recv* in *grd_comm*

with other operations is smaller than overlapping collective operations. Further, the transpose of the blocks of D^2 in line 6 can be eliminated by using a new distribution scheme for these blocks. Therefore we first eliminate line 6 in Algorithm 8 and move line 5 to the second to last line, which gives us Algorithm 9, the baseline algorithm.

Algorithm 9 is a better candidate for pipelining and overlapping communication: lines 1-2 are collective operations that can be pipelined and overlapped, lines 4-5 are collective operations that can be pipelined and overlapped, and the collective operation in line 7 can be pipelined and overlapped with two point-to-point operations in lines 8-9. Pipelining and overlapping these operations gives us Algorithm 10, the optimized SymmSquareCube algorithm. When $NDUP = 1$, the optimized algorithm is the same as the baseline algorithm.

6.2.2 Using the Multiple PPN Overlap Technique

In GTFock, the HF calculation has two major parts: Fock matrix construction and density matrix purification. The SymmSquareCube kernel is the major part of density matrix purification. To use multiple PPN overlap, we modified GTFock to allow the user to separately choose the number of MPI processes for Fock matrix construction and for density matrix purification, as described at the end of Section 6.1.2.

Algorithm 10 Optimized SymmSquareCube algorithm

Input: D and $NDUP$ copies of: *row_comm*, *col_comm* and *grd_comm*

Output: D^2, D^3 distributed as D

1: **for** $c = 1$ to $NDUP$ **do**
2: $P_{i,j,1}$ posts the broadcast of c-th part of $D_{i,j}$ as $A_{i,j}$ to $P_{i,j,:}$ using *MPI_Ibcast* with c-th *grd_comm*
3: **end for**
4: **for** $c = 1$ to $NDUP$ **do**
5: $P_{i,j,i}$ receives c-th part of $D_{i,j}$ using *MPI_Ibcast* in c-th *grd_comm*
6: $P_{i,j,i}$ posts the broadcast of c-th part of $D_{i,j}$ as $B^{\mathrm{T}}_{j,i}$ to $P_{:,j,i}$ using *MPI_Ibcast* with c-th *row_comm*
7: **end for**
8: Wait for all outstanding *MPI_Ibcast* in lines 2 and 6 to finish
9: Local matrix multiplication: $C_{i,j,k} := A_{i,j} \times B_{j,k}$
10: **for** $c = 1$ to $NDUP$ **do**
11: All processes post the reduction sum of c-th part of $C_{i,:,k}$ to c-th part of $D^2_{i,k}$ on $P_{i,i,k}$ using *MPI_Ireduce* in c-th *col_comm*
12: **end for**
13: **for** $c = 1$ to $NDUP$ **do**
14: $P_{j,j,k}$ obtains c-th part of $D^2_{j,k}$ using *MPI_Ireduce* in c-th *col_comm*
15: $P_{j,j,k}$ posts the broadcast of c-th part of $D^2_{j,k}$ as $B_{j,k}$ to $P_{:,j,k}$ using *MPI_Ibcast* with c-th *row_comm*
16: **end for**
17: Wait for all outstanding *MPI_Ibcast* in line 15 to finish
18: Local matrix multiplication: $C_{i,j,k} := A_{i,j} \times B_{j,k}$
19: **for** $c = 1$ to $NDUP$ **do**
20: All processes post the reduction sum of c-th part of $C_{i,:,k}$ to c-th part of $D^3_{i,k}$ on $P_{i,k,k}$ using *MPI_Ireduce* in c-th *col_comm*
21: **end for**
22: **for** $c = 1$ to $NDUP$ **do**
23: $P_{i,i,k}$ posts the send of c-th part of $D^2_{i,k}$ to $P_{i,k,1}$ using *MPI_Isend* and *MPI_Irecv* in c-th *global_comm*
24: $P_{i,k,k}$ waits for the c-th part of $D^3_{i,k}$ to be reduced
25: $P_{i,k,k}$ posts the send of c-th part of $D^3_{i,k}$ to $P_{i,k,1}$ using *MPI_Isend* and *MPI_Irecv* in c-th *grd_comm*
26: **end for**
27: Wait for all outstanding *MPI_Irecv* in lines 23 and 25

CHAPTER 7

PERFORMANCE TEST CALCULATIONS

To demonstrate the effect of our optimizations, we implemented all optimizations in this thesis in the GTFock code and performed test calculations.

GTFock originally used an optimized version of the ERD integral library, called OptERD [4], which has a $2\times$ performance advantage over the original ERD library [57] developed for the ACES III quantum chemistry package [29]. ERD and OptERD use the Rys quadrature [58] method for computing ERI. We optimized Simint and updated GTFock to use Simint to compute ERIs in vectorized fashion. Our implementation calls Simint one shell quartet at a time, or with batches of shell quartets using Algorithm 1. For the Fock matrix accumulation procedure, besides the original algorithm (Algorithm 2), we also implemented two optimized algorithms: Algorithm 4 and 5. Finally, we implemented Algorithm 8, 9, and 10 for the SymmSquareCube kernel in density matrix purification.

For ERI calculations, the tolerance for screening of shell quartets implemented in GTFock is 10^{-11}. The tolerance for primitive integral screening is 10^{-14} used for Simint and OptERD. These are commonly used values for these parameters.

We report the execution time spent in Fock matrix construction ("Fock build"), which includes ERI calculations ("ERI calc") and Fock matrix accumulation ("Fock accum"), and the execution time spent in density matrix purification ("Purif") as well as an SCF iteration ("SCF iter"). We also report the measured floating point operation performance (in TFlops) of the SymmSquareCube kernel to give an intuitive impression. All reported timings and performance are averaged over the SCF iterations needed for a HF-SCF calculation.

Tests were performed using the Intel Xeon Skylake (SKX) nodes on the Stampede2 supercomputer at Texas Advanced Computing Center. Each of these nodes has two sockets and 192 GB DDR4 memory, and each socket has an Intel Xeon Platinum 8160 processor

with 24 cores and 2 hyperthreads per core. The interconnect system of Stampede2 is a 100 Gbps Intel Omni-Path network with a fat tree topology employing six core switches. Codes were compiled with Intel C/C++ compiler and Intel MPI version 17.0.3 with optimization flags "-xHost -O3". Intel MKL version 17.0.3 was used to perform dense matrix-matrix multiplication in the SymmSquareCube kernel.

Tests in Section 7.2 were performed on a single SKX node using 8 MPI processes and 12 OpenMP threads per process (total of 96 threads). Tests in Section 7.3, 7.4, and 7.5 (except Table 7.12 and 7.13) were performed on 64 SKX nodes used 2 MPI processes per node and 48 OpenMP threads per process (total of 96 threads per node). Tests were performed using basis sets with different amount of contraction:

- aug-cc-pVTZ: A lightly contracted basis set,
- cc-pVDZ: A moderately contracted basis set that has few high AM shells,
- ANO-DZ: A heavily contracted basis set.

The test molecular systems are derived from a protein-ligand complex consisting of a human immunodeficiency virus (HIV) drug molecule bound to HIV II protease. The atomic configuration comes from the protein data bank (code 1HSG). Small test systems, called protein-28, consist of just the binding pocket portion of the protein. Larger test systems, called 1hsg-60 and 1hsg-70 consist of the drug molecule and a portion of its protein environment. See Table 7.1 for additional details of these test systems. Tests in Sections 7.1, 7.2, and 7.3 calculate protein-28 molecular systems using a single SKX node. Tests in Section 7.1, 7.3, 7.4, and 7.5 calculae 1hsg-60 and 1hsg-70 molecular systems using 64 SKX nodes.

7.1 Overall Results

For overall Fock matrix construction and density matrix purification performance, we compare two configurations listed in Table 7.2. We note that in the optimized version, optimiza-

Table 7.1: Test molecular systems

Test System	*Basis Set*	*Atoms*	*Occupied Orbitals*	*Shells*	*Basis Functions*
protein-28	aug-cc-pVTZ	30	62	350	1230
protein-28	cc-pVDZ	30	62	138	310
protein-28	ANO-DZ	30	62	214	526
1hsg-60	cc-pVDZ	713	1279	3138	6895
1hsg-70	cc-pVDZ	791	1418	3480	7645

Table 7.2: GTFock configurations for overall performance comparison

Configuration	*ERI Library*	*ERI Batching*	*Fock accum Algorithm*	*PGAS Framework*	*SymmSquareCube Algorithm*
Baseline	Simint	No	Alg. 2	GA v5.3	Alg. 8
Optimized	Simint	Yes	Alg. 5	GTMatrix	Alg. 10

tions for shared-memory Fock matrix accumulation in Section 4.2 and 4.3 are also enabled.

Table 7.3 shows the timings for one SCF iteration when different test molecular systems are used. Overall, we observe that compared to the baseline version, the optimized version of GTFock gives a large speedup to Fock matrix construction when using the *aug-cc-pVTZ* and *cc-pVDZ* basis sets and gives a large speedup to the density matrix purification in 64 node test cases (1hsg-60 and 1hsg-70). In the rest of this chapter, we will breakdown the timings and show the effect of each optimization seperately.

Table 7.3: Timings (in seconds) of Fock matrix construction ("Fock build"), density matrix purification ("Purif") and SCF iteration ("SCF iter") using the baseline and optimized GTFock.

Test System	*Basis Set*	*Baseline*			*Optimized*		
		Fock build	*Purif*	*SCF iter*	*Fock build*	*Purif*	*SCF iter*
protein-28	aug-cc-pVTZ	117.76	2.16	119.95	31.19	1.49	32.74
protein-28	cc-pVDZ	0.572	0.031	0.613	0.277	0.024	0.306
protein-28	ANO-DZ	193.11	0.178	193.29	201.93	0.136	202.07
1hsg-60	cc-pVDZ	37.21	4.04	41.67	12.20	2.58	15.33
1hsg-70	cc-pVDZ	45.89	4.82	51.23	15.28	3.87	19.70

Table 7.4: ERI calculation timings (in seconds) for protein-28 molecular system

Basis Set	*Scalar Simint w/o Batching*	*Vectorized Simint w/o Batching*	*Vectorized Simint w/ Batching*
aug-cc-pVTZ	53.41	47.08	15.67
cc-pVDZ	0.281	0.256	0.127
ANO-DZ	693.40	180.68	186.44

7.2 Effect of ERI Vectorization Optimizations

For ERI calculation performance, four cases are of primary interest for comparison. These are the implementations using (1) OptERD, (2) scalar Simint without batching, (3) vectorized Simint without batching, (4) vectorized Simint with batching. Scalar Simint is a version of Simint compiled without vector instructions and is useful to quantify the effectiveness of Simint's vectorization. Table 7.4 shows the timing results (in second) of all four cases. In all cases, low-level optimizations and primitive sorting are enabled in Simint.

Overall, we observe that vectorized Simint with ERI batching gives a large speedup compared to vectorized Simint without ERI batching. A surprising result is that, without batching, vectorized Simint is only slightly better than scalar Simint for the moderately contracted basis set cc-pVDZ and the lightly contracted basis set aug-cc-pVTZ. This is due to extremely short SIMD loop lengths in these two cases when batching is not used, as will be shown below. This poor performance was not noticed in the original Simint paper [5] which only used a microbenchmark to measure ERI calculation time in a way that is divorced from how Simint would actually be called from a quantum chemistry code.

For highly contracted basis sets like ANO-DZ, Simint has a large speedup of approximately 3.85$\times$ due to vectorization even without batching. For highly contracted basis sets, the vast majority of the computation is spent on well-vectorized VRRs rather than the poorly-vectorized HRRs that take place after the primitive integrals are contracted. The overhead of gathering shell pair data for batching actually has a small negative impact on ERI calulcation performance in this case.

Table 7.5: Average SIMD loop length for each call to Simint, with and without batching

Basis Set	*w/o Batching*	*w/ Batching*	*Ratio*
aug-cc-pVTZ	2.7	40.0	14.81
cc-pVDZ	7.4	71.5	9.66
ANO-DZ	79.3	1184.8	14.94

Table 7.6: ERI calculation timings (in seconds) with different queue lengths for the protein-28 molecular system

Basis Set	*32*	*24*	*16*	*12*	*8*	*4*
aug-cc-pVTZ	15.25	15.58	15.67	15.80	16.25	19.70
cc-pVDZ	0.124	0.125	0.127	0.130	0.133	0.147

For the lightly and moderately contracted basis sets, aug-cc-pVTZ and cc-pVDZ, the speedup of using vectorized Simint with batching vs. without batching is significant: 3.00 and 2.02, respectively. Therefore, batching for Simint is essential when using lightly and moderately contracted basis sets.

To support these observed results of batching on vectorization efficiency, we compare the average length of the SIMD loop in Simint with and without ERI batching. Table 7.5 shows that without batching, the average SIMD loop length when using aug-cc-pVTZ and cc-pVDZ basis sets is very small, which means that the SIMD utilization is low. Very low SIMD utilization can make vectorized Simint slower than scalar Simint, as we saw above. When using the ANO-DZ basis set, the average SIMD loop length before batching is already large enough to obtain high vectorization efficiency.

Table 7.6 shows ERI calculation timings with different queue lengths used in Algorithm 1 for the protein-28 molecular system. We skip the *ANO-DZ* basis set since already know that ERI batching has a small impact on ERI calculation performance when using the *ANO-DZ* basis set. The results justify our choice of 16 for the queue length, with longer lengths not giving significant performance improvement at the cost of additional storage.

Table 7.7 shows the effect of Simint low-level optimizations. The percentage of primitive integrals that can be neglected, and the percentage of SIMD words that can be neglected, with and without primitive sorting in Simint, is showed in Table 7.7. The speedup

Table 7.7: Effect of Simint low-level optimizations for the protein-28 molecular system

Basis Set	*Percent Neglected Primitives*	*Percent Neglected SIMD Words*		*ERI Calc. Speedup via Low-level Optimizations*
		w/o Sorting	*w/ Sorting*	
aug-cc-pVTZ	36.2	19.6	25.0	1.11
cc-pVDZ	62.0	41.6	51.3	1.17
ANO-DZ	91.3	64.6	78.4	1.41

of ERI calculations from all Simint low-level optimizations is also listed in Table 7.7. In these tests, the ERI calculation is batched.

We observe some speedup to the ERI calculation time in each case. The speedup for the aug-cc-pVTZ basis set mainly reflects the effect of optimizing for high AM functions, since aug-cc-pVTZ is lightly contracted but has many such high AM functions. The speedup for the cc-pVDZ mainly reflects and the speedup for the ANO-DZ basis sets partly reflect the effect of optimizing the contraction operation for AVX-512, given that these two basis sets have few high AM integrals, ANO-DZ is highly contracted, and the test molecule with cc-pVDZ has many integrals that belong to the $(ss|ss)$ AM class. Compared to no sorting, primitive sorting significantly reduce the number of SIMD words that need to be computed when using the ANO-DZ basis set. Therefore, the speedup for the ANO-DZ basis set mainly reflects the effect of primitive sorting.

7.3 Effect of Shared-memory Fock Matrix Accumulation Optimizations

Table 7.8 shows the average runtime of Fock matrix accumulation using Algorithm 2, 4, and 5. Optimizations in Section 4.2 and 4.3 is enabled only in Algorithm 5. The runtime for batched ERI calculation with Simint low-level optimizations is also shown for comparison. We observe that the runtime of unoptimized Fock matrix accumulation (Algorithm 2) when using aug-cc-pVTZ and cc-pVDZ basis sets is larger than the runtime of batched ERI calculation. Therefore, reducing the runtime of Fock matrix accumulation is essential.

The optimized Fock accumulation algorithm greatly reduces the runtime of Fock matrix

Table 7.8: Fock matrix accumulation ("Fock accum") timings (in seconds). Batched ERI calculation timings (in seconds) are also shown for comparison.

Test System	*Basis Set*	*Fock accum*			*ERI*
		Alg. 2	*Alg. 4*	*Alg. 5*	*Calculation*
protein-28	aug-cc-pVTZ	66.48	15.48	10.61	15.67
protein-28	cc-pVDZ	0.221	0.086	0.062	0.121
protein-28	ANO-DZ	5.29	2.20	1.51	186.44
1hsg-60	cc-pVDZ	14.62	5.90	3.78	5.48
1hsg-70	cc-pVDZ	18.61	7.60	4.82	7.46

accumulation for all tested basis sets and helps reduce Fock matrix accumulation runtime. The speedup of Algorithm 4 to Algorithm 2 shows the effect of reducing atomic operations. The speedup of Algorithm 5 to Algorithm 2 shows the effect of combining all optimizations for Fock matrix accumulation. For the ANO-DZ basis set, ERI calculation dominates the runtime of Fock build, and optimizing Fock matrix accumulation only has a small impact on Fock matrix construction runtime.

7.4 Effect of Using GTMatrix

Table 7.9 shows the runtime of the three communication procedures listed in Section 5.3 for Fock matrix construction using Global Arrays and GTMatrix. GTFock uses non-blocking calls in Global Arrays. For GTMatrix, we tested both using single blocking access and batched access. When using single blocking access in GTMatrix, its performance is comparable to Global Arrays. Using batched access in GTMatrix gives a large speedup to GatherD and ScatterF compared to using single blocking access in GTMatrix and using Global Arrays. The speedup of GatherD shows the effect of GTMatrix's ring algorithm, and the speedup of ScatterF mainly shows that batched access can greatly reduce the synchronization cost when a process has many accesses to another process. Using GTMatrix also gives a speedup to AccFBuf compared to using Global Arrays. A possible reason is that GTMatrix has a smaller process overhead.

Table 7.9: Communication procedures timing (in seconds) in Fock matrix construction using non-blocking Global Arrays operations and different access modes in GTMatrix. See Section 5.3 for explanation of communication procedure names.

Test System	*Comm. Procedure*	*Global Arrays*	*GTMatrix*	
			Blocking	*Batched*
1hsg-60	GatherD	0.473	0.478	0.330
	AccFBuf	0.183	0.133	0.134
	ScatterF	0.496	0.546	0.332
	Total	1.152	1.157	0.796
1hsg-70	GatherD	0.613	0.743	0.422
	AccFBuf	0.237	0.166	0.162
	ScatterF	0.603	0.655	0.405
	Total	1.453	1.564	0.989

Table 7.10: Performance of SymmSquareCube algorithms and speedup of the ovlpcomm algorithm (Alg. 10) over the baseline algorithm (Alg. 9)

Test System	*Matrix Dimension*	*Performance (TFlops)*			*Alg. 10 over Alg. 9*
		Alg. 8	*Alg. 9*	*Alg. 10*	
1hsg-60	6895	16.83	17.57	20.57	1.17
1hsg-70	7645	18.49	19.21	22.48	1.17

7.5 Effect of Overlapping Communication with Communications

We now compare the performance of Algorithm 8 ("original"), 9 ("baseline"), and 10 ("ovlpcomm") for the SymmSquareCube kernel. Table 7.10 shows the performance of these algorithms and the speedup of the ovlpcomm algorithm over the baseline algorithm. Overall, we observe that the baseline algorithm (Alg. 9) gives some speedup over the original algorithm (Alg. 8). Pipelining and overlapping communication operations with other communication operations in the ovlpcomm algorithm (Alg. 10) give 17% or more performance improvement over the baseline algorithm. As the baseline algorithm is already substantially optimized [4], the observed speedups are very significant.

Table 7.11 shows the performance of the ovlpcomm SymmSquareCube algorithm for different values of $NDUP$. The results justify our choice of using $NDUP = 4$. Larger $NDUP$ values can give further performance improvement, but the performance is close to that for $NDUP = 4$.

Table 7.11: Performance of ovlpcomm SymmSquareCube algorithm for different values of $NDUP$. $NDUP = 1$ is the same as the baseline SymmSquareCube algorithm.

Test	*Performance (TFlops) as function of NDUP*					
System	*1*	*2*	*3*	*4*	*5*	*6*
1hsg-60	17.57	19.82	19.43	20.57	21.21	20.68
1hsg-70	19.21	21.51	21.47	22.48	22.39	22.54

Table 7.12 shows the performance of the ovlpcomm SymmSquareCube algorithm with $NDUP = 1$ and $NDUP = 4$ using different numbers of MPI processes per node (PPN) for the 1hsg-70 molecular system. The case $NDUP = 1$ corresponds to the baseline algorithm without the nonblocking overlap technique, and thus shows the effect of the multiple PPN overlap technique by itself.

The number of MPI processes per node, PPN, is chosen such that $64 \times (PPN - 1) < p^3 \leq 64 \times PPN$, where p^3 is the number of processes. The column "total nodes" is the actual number of nodes utilized, $\lceil p^3/PPN \rceil$. We use a "natural" assignment of the MPI ranks to the $p \times p \times p$ process mesh, i.e., the ranks are assigned row by row in one plane and then plane by plane. Also, the MPI ranks on a node are numbered consecutively.

We observe that using multiple MPI processes per node gives considerable speedup to the SymmSquareCube algorithm with either $NDUP = 1$ or $NDUP = 4$ compared to using a single MPI process per node. When running multiple MPI processes per node, using $NDUP = 4$ is always faster than using $NDUP = 1$. It is surprising that, for the ovlpcomm SymmSquareCube algorithm, using $NDUP = 4$ with only 2 MPI processes per node is almost always faster than using $NDUP = 1$ with any number of MPI processes per node. This shows that combining the two techniques, nonblocking overlap and multiple PPN overlap, is a better choice than using only one of the techniques. The best performance of SymmSquareCube, combining the two overlapping techniques ($7 \times 7 \times 7$ processes with $NDUP = 4$), is 91.2% faster than the baseline performance without use of communication overlap.

However, using a large number of PPN may not give the best overall performance.

Table 7.12: Performance of the ovlpcomm SymmSquareCube algorithm with $NDUP = 1$ and 4 for different numbers of PPN. Test molecular system is 1hsg-70.

Process Configuration			*SymmSquareCube Performance (TFlops)*	
PPN	*Process Mesh*	*Total Nodes*	$NDUP = 1$	$NDUP = 4$
1	$4 \times 4 \times 4$	64	19.21	22.48
2	$5 \times 5 \times 5$	63	20.61	26.45
4	$6 \times 6 \times 6$	54	26.24	33.87
6	$7 \times 7 \times 7$	58	27.53	36.73
8	$8 \times 8 \times 8$	64	24.98	32.38

Table 7.13: Timings (in seconds) of Fock matrix construction ("Fock build"), density matrix purification ("Purif") and SCF iteration ("SCF iter") using $NDUP = 4$ for different numbers of PPN. Test molecular system is 1hsg-70.

PPN	*Fock build*	*Purif*	*SCF iter*
1	17.26	3.52	21.25
2	15.13	3.23	18.60
4	17.26	2.53	20.09
6	18.54	2.52	21.33
8	19.51	2.86	22.67

Table 7.13 shows the runtime of Fock matrix construction, density matrix purification, and SCF iteration using different numbers of MPI PPN for the 1hsg-70 molecular system and $NDUP = 4$. We can see that using PPN=6 does not give the best SCF performance. Instead, using PPN=2 and a $5 \times 5 \times 5$ process mesh gives the best SCF performance.

CHAPTER 8
CONCLUSION

In this thesis, we have described and demonstrated several performance optimizations for computational kernels in quantum chemistry calculations. Chapter 3 introduced optimizations to accelerate ERI calculations utilizing a processor's vector processing units: low-level optimizations for the Simint library and an ERI batching scheme for Fock matrix construction. Chapter 4 discussed the steps to reduce the cost of thread-safe parallel Fock matrix accumlation on shared-memory platforms and other techniques to improve the performance of Fock matrix accumulation. Chapter 5 presented the design of GTMatrix, a new portable PGAS framework, and the usage of this library. Chapter 6 explored a new idea of overlapping communications with communications, which may better utilize network bandwidth and speed up communication intensive kernels like the matrix squaring and cubing kernel in density matrix purification. Test calculations in Chapter 7 show that the optimizations in this thesis give favourable speedups to the optimized kernels, which justifies the optimizations in this thesis. GTFock is released in open-source form at `https://github.com/gtfock-chem`.

REFERENCES

[1] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. Dover Publications, 2006.

[2] J. A. Pople, P. M. Gill, and B. G. Johnson, "Kohn-Sham density-functional theory within a finite basis set," *Chemical Physics Letters*, vol. 199, no. 6, pp. 557–560, 1992.

[3] H. Shan, B. Austin, W. De Jong, L. Oliker, N. Wright, and E. Aprà, "Performance tuning of Fock matrix and two-electron integral calculations for NWChem on leading HPC platforms," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS13) held as part of SC13*, 2013.

[4] E. Chow, X. Liu, S. Misra, M. Dukhan, M. Smelyanskiy, J. R. Hammond, Y. Du, X.-K. Liao, and P. Dubey, "Scaling up Hartree-Fock calculations on Tianhe-2," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 85–102, 2015.

[5] B. P. Pritchard and E. Chow, "Horizontal vectorization of electron repulsion integrals," *Journal of Computational Chemistry*, vol. 37, no. 28, pp. 2537–2546, 2016.

[6] H. Huang and E. Chow, "Accelerating quantum chemistry with vectorized and batched integrals," in *SC 18': Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, Texas, Nov. 2018.

[7] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the Global Arrays shared memory programming toolkit," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.

[8] R. McWeeny, "Some recent advances in density matrix theory," *Reviews of Modern Physics*, vol. 32, no. 2, pp. 335–369, 1960.

[9] D. R. Bowler and T Miyazaki, "O(n) methods in electronic structure calculations," *Reports on Progress in Physics*, vol. 75, no. 3, p. 036 503, 2012.

[10] E. Chow, X. Liu, M. Smelyanskiy, and J. R. Hammond, "Parallel scalability of Hartree-Fock calculations," *The Journal of Chemical Physics*, vol. 142, no. 10, p. 104 103, 2015.

[11] X. Liu, A. Patel, and E. Chow, "A new scalable parallel algorithm for Fock matrix construction," *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

[12] R. J. Harrison, G. Beylkin, F. A. Bischoff, J. A. Calvin, G. I. Fann, J. Fosso-Tande, D. Galindo, J. R. Hammond, R. Hartman-Baker, J. C. Hill, J. Jia, J. S. Kottmann, M.-J. Yvonne Ou, J. Pei, L. E. Ratcliff, M. G. Reuter, A. C. Richie-Halford, N. A. Romero, H. Sekino, W. A. Shelton, B. E. Sundahl, W. S. Thornton, E. F. Valeev, Álvaro Vázquez-Mayagoitia, N. Vence, T. Yanai, and Y. Yokoi, "MADNESS: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, S123–S142, 2016.

[13] V. R. Saunders and M. F. Guest, "Applications of the CRAY-1 for quantum chemistry calculations," *Computer Physics Communications*, vol. 26, no. 3, pp. 389 –395, 1982.

[14] P. M. W. Gill, M. Head-Gordon, and J. A. Pople, "Efficient computation of two-electron - repulsion integrals and their nth-order derivatives using contracted Gaussian basis sets," *The Journal of Physical Chemistry*, vol. 94, no. 14, pp. 5564–5572, 1990.

[15] K. Wolinski, R. Haacke, J. F. Hinton, and P. Pulay, "Methods for parallel computation of SCF NMR chemical shifts by GIAO method: Efficient integral calculation, multi-Fock algorithm, and pseudodiagonalization," *Journal of Computational Chemistry*, vol. 18, no. 6, pp. 816–825, 1997.

[16] H. Shan, S. Williams, W. de Jong, and L. Oliker, "Thread-level parallelization and optimization of NWChem for the Intel MIC architecture," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '15, San Francisco, California: ACM, 2015, pp. 58–67.

[17] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. Van Dam, D. Wang, J. Nieplocha, E. Aprà, T. Windus, and et al., "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[18] E. F. Valeev, *A library for the evaluation of molecular integrals of many-body operators over Gaussian functions*, http://libint.valeyev.net/, 2014.

[19] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery, "General atomic and molecular electronic structure system," *Journal of Computational Chemistry*, vol. 14, no. 11, pp. 1347–1363,

[20] K. Yasuda, "Two-electron integral evaluation on the graphics processor unit," *Journal of Computational Chemistry*, vol. 29, no. 3, pp. 334–342, 2007.

[21] I. S. Ufimtsev and T. J. Martinez, "Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation," *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, 2008.

[22] A. Asadchev, V. Allada, J. Felder, B. M. Bode, M. S. Gordon, and T. L. Windus, "Uncontracted Rys quadrature implementation of up to g functions on graphical processing units," *Journal of Chemical Theory and Computation*, vol. 6, no. 3, pp. 696–704, 2010.

[23] N. Luehr, I. S. Ufimtsev, and T. J. Martinez, "Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs)," *Journal of Chemical Theory and Computation*, vol. 7, no. 4, pp. 949–954, 2011.

[24] K. A. Wilkinson, P. Sherwood, M. F. Guest, and K. J. Naidoo, "Acceleration of the GAMESS-UK electronic structure package on graphical processing units," *Journal of Computational Chemistry*, vol. 32, no. 10, pp. 2313–2318, 2011.

[25] Y. Miao and K. M. Merz, "Acceleration of electron repulsion integral evaluation on graphics processing units via use of recurrence relations," *Journal of Chemical Theory and Computation*, vol. 9, no. 2, pp. 965–976, 2013.

[26] T. Ramdas, G. K. Egan, D. Abramson, and K. K. Baldridge, "On ERI sorting for SIMD execution of large-scale Hartree – Fock SCF," *Computer Physics Communications*, vol. 178, no. 11, pp. 817 –834, 2008.

[27] ——, "Uniting extrinsic vectorization and shell structure for efficient simd evaluation of electron repulsion integrals," *Chemical Physics*, vol. 349, no. 1–3, pp. 147 –157, 2008, Electron Correlation and Molecular Dynamics for Excited States and Photochemistry.

[28] ——, "ERI sorting for emerging processor architectures," *Computer Physics Communications*, vol. 180, no. 8, pp. 1221 –1229, 2009.

[29] V. Lotrich, N. Flocke, M. Ponton, A. D. Yau, A. Perera, E. Deumens, and R. J. Bartlett, "Parallel implementation of electronic structure energy, gradient, and Hessian calculations," *The Journal of Chemical Physics*, vol. 128, no. 19, p. 194 104, 2008.

[30] C. L. Janssen and I. M. B. Nielsen, *Parallel Computing in Quantum Chemistry*. Taylor & Francis, 2008.

[31] J. M. Turney, A. C. Simmonett, R. M. Parrish, E. G. Hohenstein, F. A. Evangelista, J. T. Fermann, B. J. Mintz, L. A. Burns, J. J. Wilke, M. L. Abrams, N. J. Russ, M. L. Leininger, C. L. Janssen, E. T. Seidl, W. D. Allen, H. F. Schaefer, R. A. King, E. F. Valeev, C. D. Sherrill, and T. D. Crawford, "PSI4: an open-source *ab initio* electronic structure program," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 2, pp. 556–565, 2012.

[32] Q. Sun, T. C. Berkelbach, N. S. Blunt, G. H. Booth, S. Guo, Z. Li, J. Liu, J. D. McClain, E. R. Sayfutyarova, S. Sharma, S. Wouters, and G. K.-L. Chan, "PySCF: The Python-based simulations of chemistry framework," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 8, no. 1, e1340, 2018.

[33] V. Mironov, Y. Alexeev, K. Keipert, M. D'mello, A. Moskovsky, and M. S. Gordon, "An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel Xeon Phi processor," in *SC 17': Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, Colorado, Nov. 2017.

[34] R. A. van de Geijn and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1995.

[35] E. Dekel, D. Nassimi, and S. Sahni, "Parallel matrix and graph algorithms," *SIAM Journal on Computing*, vol. 10, no. 4, pp. 657–675, 1981.

[36] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[37] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," *Euro-Par 2011 Parallel Processing*, pp. 90–109, 2011.

[38] A. Buluc and J. R. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ser. ICPP '08, IEEE Computer Society, 2008, pp. 503–510, ISBN: 978-0-7695-3374-2.

[39] ——, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[40] J. A. Calvin, C. A. Lewis, and E. F. Valeev, "Scalable task-based algorithm for multiplication of block-rank-sparse matrices," in *Proceedings of the 5th Workshop on*

Irregular Applications: Architectures and Algorithms, ser. IA3 '15, Austin, Texas: ACM, 2015, 4:1–4:8, ISBN: 978-1-4503-4001-4.

[41] T. Helgaker, P. Jørgensen, and J. Olsen, *Molecular Electronic-Structure Theory*. John Wiley & Sons, LTD, 2000.

[42] S. Obara and A. Saika, "General recurrence formulas for molecular integrals over Cartesian Gaussian functions," *The Journal of Chemical Physics*, vol. 89, no. 3, pp. 1540–1559, 1988.

[43] ——, "Efficient recursive computation of molecular integrals over Cartesian Gaussian functions," *The Journal of Chemical Physics*, vol. 84, no. 7, pp. 3963–3974, 1986.

[44] R. Lindh, U. Ryu, and B. Liu, "The reduced multiplication scheme of the Rys quadrature and new recurrence relations for auxiliary function based two-electron integral evaluation," *The Journal of Chemical Physics*, vol. 95, no. 8, pp. 5889–5897, 1991.

[45] T. P. Hamilton and H. F. Schaefer, "New variations in two-electron integral evaluation in the context of direct SCF procedures," *Chemical Physics*, vol. 150, no. 2, pp. 163 –171, 1991.

[46] Colfax Research, *Capabilities of Intel AVX-512 in Intel Xeon Scalable processors (Skylake)*, `https://colfaxresearch.com/skl-avx512/`.

[47] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier, 2016.

[48] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," in *Parallel and Distributed Processing*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 533–546.

[49] J. Daily, A. Vishnu, H. van Dam, B. Palmer, and D. J. Kerbyson, "On the suitability of MPI as a PGAS runtime," in *International Conference on High Performance Computing (HiPC)*, 2014.

[50] *NWGrid home page*, `http://www.emsl.pnl.gov/nwgrid/`, Accessed: 2019-03-17.

[51] M. D. White and M. Oostrom, "STOMP: Subsurface Transport Over Multiple Phases version 4.0 user guide," Pacific Northwest National Laboratory, Richland, Washington, Tech. Rep., 2006.

[52] C. S. Oehmen and D. J. Baxter, "ScalaBLAST 2.0: Rapid and robust BLAST calculations on multiprocessor systems," *Bioinformatics*, vol. 29, no. 6, pp. 797–798, 2013.

[53] H.-J. Werner, P. J. Knowles, G. Knizia, F. R. Manby, and M. Schutz, "Molpro: A general-purpose quantum chemistry program package," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 2, no. 2, pp. 242–253,

[54] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS extension for C++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 1105–1114.

[55] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014, ISBN: 0262527634, 9780262527637.

[56] M. P. I. Forum, "MPI: A Message-Passing Interface Standard, version 3.1," University of Tennessee, Tech. Rep., 2015.

[57] N. Flocke and V. Lotrich, "Efficient electronic integrals and their generalized derivatives for object oriented implementations of electronic structure calculations," *Journal of Computational Chemistry*, vol. 29, no. 16, pp. 2722–2736, 2008.

[58] J. Rys, M. Dupuis, and H. F. King, "Computation of electron repulsion integrals using the Rys quadrature method," *Journal of Computational Chemistry*, vol. 4, no. 2, pp. 154–157, 1983.