2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

Overlapping Communications with Other Communications and its Application to Distributed Dense Matrix Computations

Hua Huang
College of Computing
Georgia Institute of Technology
Atlanta, GA, U.S.A.
huangh223@gatech.edu

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, GA, U.S.A.
echow@cc.gatech.edu

***Abstract*—This paper presents the idea of *overlapping communications with communications*. Communication operations are overlapped, allowing 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 use two techniques for overlapping communication operations: a novel technique called "nonblocking overlap" that uses MPI-3 nonblocking collective operations and software pipelines, and a simpler technique that uses multiple MPI processes per node to send different portions of data simultaneously. The idea is applied to the parallel dense matrix squaring and cubing kernel in density matrix purification, an important kernel in electronic structure calculations. The kernel is up to 91.2% faster when communication operations are overlapped.**

***Index Terms*—Message Passing Interface, nonblocking collective communication, pipelined and overlapped communications, parallel matrix multiplication, density matrix purification**

I. Introduction

In many distributed memory computations, internode communication is the bottleneck, even when communication-optimal algorithms are used. In these cases, it is important to ensure that the network bandwidth is utilized as fully as possible. The network bandwidth may not be fully utilized due to (a) the overhead of process synchronization in two-sided communication protocols, (b) the overhead of data marshalling and possibly data copying in the message passing library, and (c) computations that need to be performed during a communication operation, such as those in reduction operations. Indeed, when short messages must be sent, network utilization is poor, due to various overheads lumped under the term "latency."

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. The technique has similarities to the idea of overlapping communications with computations, allowing a program to exploit the concurrency between communication and computation units. However, in communication-intensive programs, the computation time is small compared to the communication time, and thus the benefit of overlapping communications with computations can be very limited.

We consider two techniques for overlapping communication operations:

1) We introduce the technique of dividing the data and sending it via separate nonblocking MPI calls within one thread of an MPI process. In particular, the nonblocking collective operations [1] provided by the MPI-3 standard allows us to overlap collectives using this technique (although they were designed to provide overlapping of collective operations with computations). We find that this overlapping technique can work well if different types of communication can be overlapped (e.g., a nonblocking broadcast with nonblocking send-receive) and if the communication of the divided parts of the data can be pipelined. We call this technique "nonblocking overlap."
2) A simpler technique is to use multiple MPI processes per node (PPN) where each process communicates a portion of the data that would be sent when using a single MPI process per node. This technique has been used in the past in cases where a single MPI process on a node cannot saturate that node's available network bandwidth. However, this technique can increase the required communication bandwidth, so it is not obvious that using multiple PPN can reduce overall time. Distributed memory programs usually use the same number of PPN for all parts of the program, but we argue that different parts of a program could use different numbers of PPN for best performance. We call this technique "multiple PPN overlap."

We remark in passing that a third technique for overlapping communication operations is to use multithreading. Here, multiple threads within one process will call MPI functions. Unfortunately, this technique usually has high overheads due

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00060

to the need to guarantee thread safety within multithreaded MPI, in addition to the overhead of multithreading itself. Our tests with using multithreading to overlap communication operations typically show poor performance (particularly for message sizes less than 64K) compared to using the above two techniques. We do not further consider multithreading for overlapping communications in this paper.

The two techniques (1) and (2) above can also be combined, i.e., using multiple MPI processes per node where each process also overlaps its communications using nonblocking calls. Combining the two techniques gives the programmer flexibility in controlling how many processes to use, how much overlap to use (number of communication operations that are called simultaneously), and on what nodes the processes reside. Our results will show that combining the two techniques appears to give the best performance results.

The nonblocking overlap technique can be introduced unobtrusively to existing application codes. In many cases, an "optimal" number of PPN may already have been chosen, for example, based on memory capacity or performance. This value of PPN does not need to be changed when using nonblocking overlap.

The multiple PPN overlap technique implies that the number of PPN is chosen to optimize communication performance. A different number of PPN may be best for different parts of an application code. In this case, the code would need the capability to create multiple processes per node and to use just the right number of processes per node when needed (processes sleep when they are not needed).

In this paper, we demonstrate these two techniques of overlapping communication with communication using a distributed dense matrix computation. These computations are dominated by communication; the on-node computations (DGEMMs) require comparatively little time, especially compared to five years ago. These computations also often use a 2D partitioning of the matrix (or 3D or 2.5D partitioning of the work), and use collective operations along rows and columns of a processor mesh. Such collectives can be implemented in many ways, but often do not achieve the peak bandwidth available on the interconnect. This deficiency will be demonstrated in Section V, and we will also show that overlapping such communication with itself or with other types of communications can improve performance.

The dense matrix computations are applied in a kernel for computing the square and the cube of a symmetric matrix. This kernel arises in the density matrix purification algorithm in density functional theory (DFT) and other electronic structure methods. In DFT, the bottleneck is an eigendecomposition of a Hamiltonian or Fock matrix, F, used to compute a spectral projector called a density matrix, D. Instead of eigendecomposition, D can be computed directly from F by using the density matrix purification iteration [2]

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

where the initial approximation D_0 is an appropriately scaled and shifted version of F. Thus, matrix squaring and cubing are needed at each step. This iteration has many practical variations, but all the variations involve squaring and cubing, or forming even higher matrix powers. We use the "canonical purification" method [3] in our experiments. In linear scaling DFT, the density matrices are large and sparse [4]. In Hartree-Fock theory, 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 [5].

This paper makes the following contributions:

1) We promote the idea of overlapping communication with communication. We illustrate how overlap can be accomplished with the new nonblocking overlap technique. We also discuss how multiple PPN can be used as an overlap technique (Section III).
2) We develop a new implementation of the parallel matrix squaring and cubing kernel optimized with the nonblocking overlap technique (Section IV).
3) We analyze the effect of the two overlap techniques using micro-benchmarks and on the new parallel matrix squaring and cubing kernel (Section V).

II. Background and Related Work

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) [6] is the most widely used 2D algorithm.

To reduce the communication cost, 3D algorithms [7], [8] 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 [9] 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 [10], [11], 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 [12].

III. Techniques for Overlapping Communications

A. 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 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 1 is the algorithm for the parallel matrix-vector multiplication. Figure 1 illustrates the communications in the algorithm for a 4×4 process mesh.

Algorithm 1 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 1, the communicated data in lines 2-3 can be divided and the operations 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 *N_DUP* copies of *row_comm* and *col_comm*, overlapping the communications in lines 2 and 3 of Algorithm 1 with nonblocking operations gives Algorithm 2. Figure 2 shows the communications in Algorithm 2 for a 4×4 process mesh and $N\_DUP = 2$.

In Algorithm 2, 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 adjacent 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

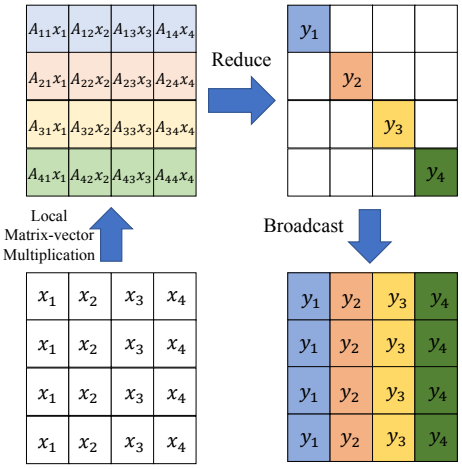

Fig. 1. Communication operations in Algorithm 1 for a 4×4 process mesh. Colors denote different blocks of the vector y.

Algorithm 2 Parallel matrix-vector multiplication with pipelined and overlapped communications

Input: A, x and *N_DUP* 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 *N_DUP* equal-size contiguous parts
3: **for** $c = 1$ to *N_DUP* **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 *N_DUP* **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

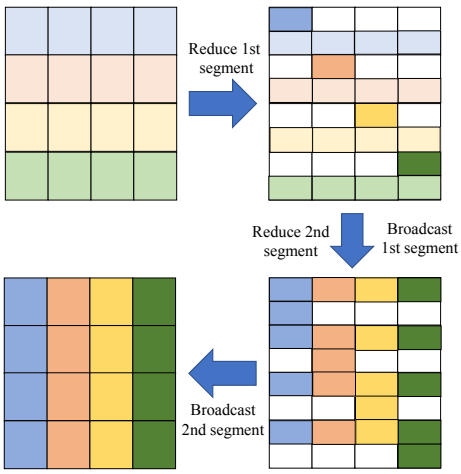

Fig. 2. Overlapped and pipelined communication operations in Algorithm 2 for a 4×4 process mesh and $N\_DUP = 2$. Colors denote different blocks of vector y.

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 N_DUP is also important. One can use different N_DUP values for different operations, if these operations are not overlapped with each other. The best N_DUP 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 $1/N\_DUP$ and N_DUP operations are issued and pipelined. The actual inter-node bandwidth may not be as high as N_DUP times $f_{BW}(\frac{n}{N\_DUP})$. To further utilize the network bandwidth,

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

is a necessary condition. An easier way to choose N_DUP is to make sure n/N_DUP 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 $n/N\_DUP \leq n_t$, 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 N_DUP (such as 16) may give some speedup over using a small or moderate value (such as 4), but using a very large value of N_DUP would heavily consume system resources and have a large overhead.

B. 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.

Thus, in order to understand if and when the multiple PPN overlap technique is effective, particularly for collectives, we test the technique with a micro-benchmark and with our density matrix purification application in Section V.

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 V.

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 paper, 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 milliseconds. 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.

IV. 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. In Section V-E, we test SymmSquareCube based on 2.5D 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 3 is the original algorithm for SymmSquareCube released in the GTFock code [5], [13], [14]. Algorithm 3 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 3. The symmetry of D is only used in line 2.

A. Using the Nonblocking Overlap Technique

Algorithm 3 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 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 3 and move line 5 to the second to last line, which gives us Algorithm 4, the baseline algorithm.

Algorithm 4 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 5, the optimized SymmSquareCube algorithm. When $N\_DUP = 1$, the optimized algorithm is the same as the baseline algorithm.

Algorithm 3 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 4 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*

B. Using the Multiple PPN Overlap Technique

In the GTFock code mentioned earlier, the Hartree-Fock calculation has two major kernels: 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 III-B.

Algorithm 5 Optimized SymmSquareCube algorithm

Input: D and N_DUP copies of: *row_comm*, *col_comm* and *grd_comm*

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

1: **for** $c = 1$ to N_DUP **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 N_DUP **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 N_DUP **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 N_DUP **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 N_DUP **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 N_DUP **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

V. Performance Results

Tests were performed with the original, baseline, and optimized SymmSquareCube algorithms (Algorithms 3, 4, and 5, respectively) for computing D^2 and D^3 required by canonical purification. For all tests, we report the average number of floating-point operations per second of SymmSquareCube. These values are averaged over all the self-consistent field (SCF) iterations needed for a Hartree-Fock calculation implemented in GTFock [5], [13], [14].

Tests were performed using the Intel Xeon Skylake 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 version 17.0.4 and Intel MPI version 17.0.3 with optimization flags "-xHost -O3". Intel MKL version 17.0.4 was used to perform local dense matrix multiplication.

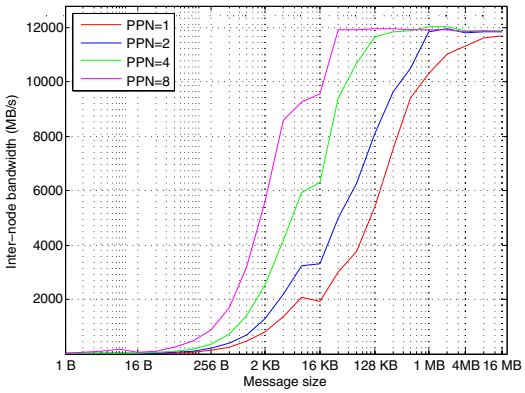

Fig. 3. Unidirectional point-to-point bandwidth versus message size with different numbers of PPN on two Stampede2 Skylake nodes.

A. Analysis of Actual Bandwidth

We first determine the achievable unidirectional bandwidth across two Stampede2 Skylake nodes using point-to-point communication with different numbers of processes per node. We put all source processes on one node and all destination processes on a second node. Figure 3 shows the measured unidirectional inter-node bandwidth versus message size for different PPN. We observe that the peak unidirectional bandwidth is about 12000 MB/s. Except for very large message sizes, the peak available bandwidth cannot be attained by a single process per node without overlapping communications. This is the root motivation for overlapping communication operations.

We model the communication time of collective operations used in SymmSquareCube. The time taken to send a message between any two nodes can be modeled as $\alpha + n\beta$, where α is the network latency, β is the transfer time per byte (the inverse of network bandwidth), and n is the message size. For collective operations, Intel MPI can automatically choose an algorithm according to parameters such as message size and number of processes [15]. For theoretical analysis, we assume the recursive doubling algorithm for broadcast and Rabenseifner's algorithm [16] for reduction because these are appropriate choices for long messages. For these algorithms,

$$T_{Bcast} = \alpha(log(p) + p - 1) + 2\beta(p-1)n/p,$$
$$T_{Reduce} = 2\alpha log(p) + 2\beta(p-1)n/p.$$

Now we consider the theoretical and actual time consumption of the baseline SymmSquareCube algorithm on 64 Skylake nodes for molecular system 1hsg_70 using a single MPI process per node. The matrix dimension for 1hsg_70 is 7645. (Details of the molecular systems are in Ref. [14] but are immaterial to this paper except for the dimension of the density matrices.) In this situation, $p^3 = 64$, $p = 4$, the largest matrix block size is $(\lceil 7645/4 \rceil)^2 = 1912^2$, and the message size is $1912^2 \times 8$ bytes = 27.89 MB. Since the message size is large, the communication time should be dominated by data transfer (the β terms), and thus we can ignore the network latency (the α terms). Substituting $p = 4$, $n = 27.89$ MB, and β = 1 / (12000 MB/s), we have the theoretical communication time for the baseline algorithm:

$$T_{P2P} = 2.324 \times 10^{-3},$$
$$T_{Bcast} = T_{Reduce} = 3.487 \times 10^{-3},$$
$$T_{baseline} = 2 \times (T_{P2P} + T_{Reduce}) + 3 \times T_{Bcast} = 0.02208.$$

However, the average communication time for the baseline algorithm is 0.07312 seconds, so the actual bandwidth is just 30.19% of the peak bandwidth. Meanwhile, two local matrix multiplications in the baseline algorithm take 0.01794 seconds on average, showing that communication indeed dominates computation.

B. Micro-benchmark

We developed a micro-benchmark to measure the actual bandwidth for broadcast and reduction operations across 4 nodes in three different cases. The second and third cases overlap communication operations.

1) Blocking: this is the standard non-overlapped case using one MPI process per node;
2) Nonblocking overlap with $N\_DUP = 4$: this is the new technique using one MPI process per node and 4 MPI communicators on each process. Each communicator performs nonblocking collective operations with $1/4$ of the original message;
3) 4 PPN overlap: this is the multiple PPN overlap technique with 4 processes for each node. Each process is involved in a blocking collective operation with processes on other nodes with $1/4$ of the original message. Four collective operations are performed in overlapped fashion.

Figure 4 shows the communication patterns of the first and third cases, and in particular justifies the configuration used in the third case. The third case has the same number of MPI processes in each communication group and the same inter-node communication volume as the first and the second cases.

Figure 5 shows the actual bandwidth of both broadcast and reduction for these cases, assuming the communication volume is $2(p-1)n/p$ for recursive doubling or Rabenseifner's algorithm, where n is the message size and $p = 4$ is the number of processes participating in the collectives. For reduction, we used the *MPI_FLOAT* data type and the *MPI_SUM* operation.

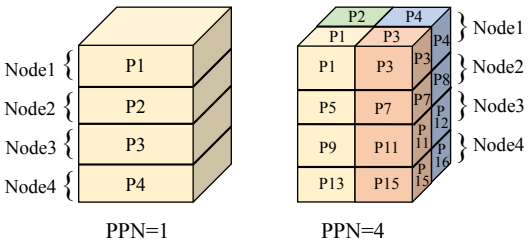

Fig. 4. Communication patterns of the blocking and 4 PPN blocking cases used in collective operation micro-benchmark. PPN=1 shows collective communication between 4 nodes and 1 process per node. Each process holds data of length N. PPN=4 shows the same effective communication using 4 nodes and 4 processes per node. Each process holds data of length $N/4$ and communicates in a column communicator consisting of 4 processes.

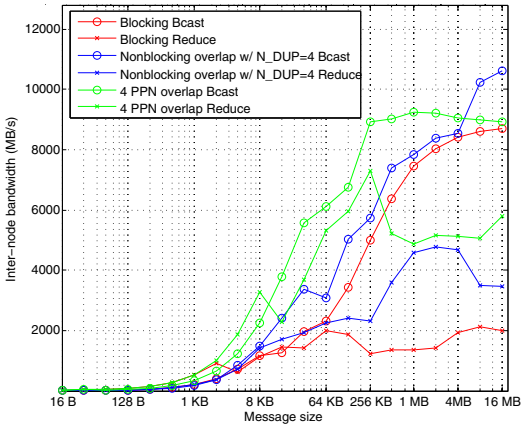

Fig. 5. Measured broadcast and reduction bandwidth versus message size on 4 nodes. Three cases are tested: blocking (not overlapped), nonblocking overlap with $N\_DUP = 4$, and 4 PPN overlap.

We observe that the actual bandwidth of the blocking broadcast is only about 75% of the peak unidirectional point-to-point bandwidth when the message size is 16 MB. The actual bandwidth of the blocking reduction is much smaller than the peak unidirectional point-to-point bandwidth, which is the main reason that the actual bandwidth of the baseline SymmSquareCube algorithm is far from the peak unidirectional point-to-point bandwidth. Both nonblocking overlap and multiple PPN overlap can improve the performance of the single blocking collective operations.

Figure 6 shows the corresponding timing data on node 0 for the 8MB case. For the nonblocking operations, the timings are split into the time for posting the operation and waiting for the operation to complete. Timings for 2MB messages using single blocking and nonblocking collective operations are also shown for comparison.

Consider first the reduction operation (top half of Figure 6). In the nonblocking overlap case, the nonblocking reductions are posted in sequence, and this is clearly seen in the figure. This serialization of the posting overhead makes the nonblocking overlap technique worse than the multiple PPN overlap technique (shown in light blue). Both techniques are much

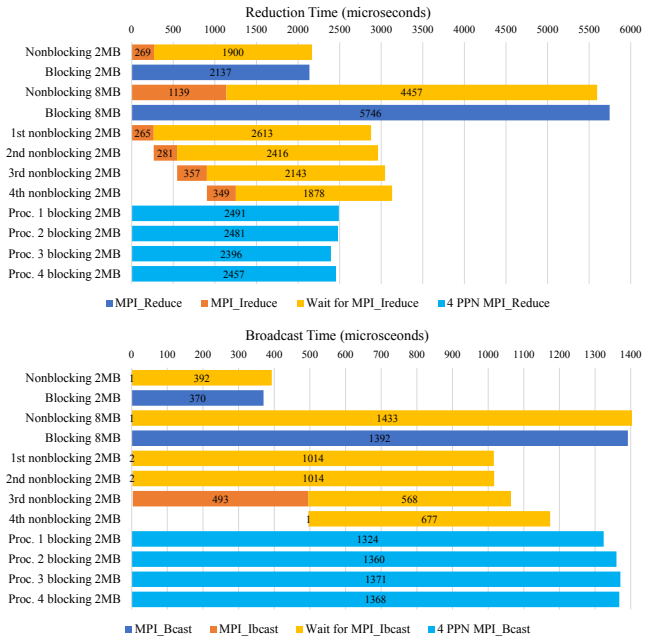

Fig. 6. Time diagram for reducing (top) and broadcasting (bottom) 8MB data using blocking (not overlapped), nonblocking overlapped with $N\_DUP = 4$, and 4 PPN overlap on process 0. Timings for broadcasting and reducing (a) 2MB data using single blocking and nonblocking calls (not overlapped), and (b) 8MB data using single nonblocking call (not overlapped), are also shown for comparison.

faster than the non-overlapped case, which is evidence that the network resources are better utilized. The data for the non-overlapped nonblocking reduction shows that time for posting the operation (1139μs) is comparable to the time for posting all four overlapped nonblocking operations ($265+281+357+349\mu$s). That all four nonblocking operations complete at approximately the same time suggests that actual data transfer may not be starting until the last nonblocking operation is posted.

The situation is different for the broadcast operation for this message size (bottom half of Figure 6). The nonblocking overlap technique is better than the multiple PPN overlap technique. Here, posting the nonblocking broadcasts in non-blocking overlap take either very little time, or a significant amount of time (e.g., 493μs). As a comparison, posting nonblocking broadcasts for multiple PPN overlap take very little time (data not shown), which suggests that actual data transfer may have started before posting the third nonblocking broadcast in the nonblocking overlap case.

C. SymmSquareCube Results for Nonblocking Overlap with $N\_DUP = 4$

We now compare the performance of the original, baseline, and optimized SymmSquareCube algorithms. Table I shows the performance of these algorithms and the speedup of the optimized algorithm over the baseline algorithm. As the baseline algorithm is already substantially optimized [14], the observed speedups are very significant.

TABLE I
PERFORMANCE OF SYMMSQUARECUBE ALGORITHMS AND SPEEDUP OF THE OPTIMIZED ALGORITHM (ALG. 5) OVER THE BASELINE ALGORITHM (ALG. 4)

Test System	*Matrix Dimension*	*Performance (TFlops)*			Alg. 5 over Alg. 4
		Alg. 3	*Alg. 4*	*Alg. 5*	
1hsg_45	5330	12.36	13.20	16.05	1.21
1hsg_60	6895	16.83	17.57	20.57	1.17
1hsg_70	7645	18.49	19.21	22.48	1.17

TABLE II
PERFORMANCE OF OPTIMIZED SYMMSQUARECUBE FOR DIFFERENT VALUES OF N_DUP. $N\_DUP = 1$ IS THE SAME AS THE BASELINE SYMMSQUARECUBE ALGORITHM.

Test System	*Performance (TFlops) as function of N_DUP*					
	1	*2*	*3*	*4*	*5*	*6*
1hsg_45	13.17	15.30	14.61	16.05	16.19	16.07
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

Overall, we observe that the baseline algorithm (Alg. 4) gives some speedup over the original algorithm (Alg. 3). Pipelining and overlapping communication operations with other communication operations in the optimized algorithm (Alg. 5) give 17% or more performance improvement over the baseline algorithm. Transposing D^2 (line 6) in the original algorithm has fewer steps and smaller communication volume compared to the collective operations, so we can expect that the speedup of the baseline algorithm over the original algorithm to be smaller when using more MPI processes. For smaller matrices, the synchronization cost relative to data transfer is larger. Therefore, pipelining and overlapping communications and adjusting point-to-point operations give larger speedup for 1hsg_45 compared to 1hsg_60 and 1hsg_70. For larger matrices, pipelining and overlapping communications benefit more from better utilizing communication bandwidth.

Table II shows the performance of the optimized SymmSquareCube algorithm for different values of N_DUP. The results justify our choice of using $N\_DUP = 4$. Larger N_DUP values can give further performance improvement, but the performance is close to that for $N\_DUP = 4$.

D. SymmSquareCube Results for Multiple PPN Overlap

Table III shows the performance of the optimized SymmSquareCube algorithm with $N\_DUP = 1$ and $N\_DUP = 4$ using different numbers of MPI processes per node for the 1hsg_70 molecular system. The case $N\_DUP = 1$ 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,

TABLE III
PERFORMANCE OF THE OPTIMIZED SYMMSQUARECUBE ALGORITHM WITH $N\_DUP = 1$ AND 4 FOR DIFFERENT NUMBERS OF PPN

Process Configuration			*Performance (TFlops)*	
PPN	*Process Mesh*	*Total Nodes*	$N\_DUP = 1$	$N\_DUP = 4$
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 IV
ESTIMATED INTER-NODE COMMUNICATION VOLUME, BANDWIDTH, AND TIME USING MICRO-BENCHMARK DATA AND ACTUAL INTER-NODE COMMUNICATION TIME OF THE BASELINE SYMMSQUARECUBE ALGORITHM FOR DIFFERENT NUMBERS OF PPN

	Estimated inter-node communication				*Actual*
PPN	*volume (MB)*	*Reduce BW (GB/s)*	*Bcast BW (GB/s)*	*time (s)*	*inter-node comm. time (s)*
1	265.0	2.4	8.5	0.058	0.073
2	311.5	3.1	8.8	0.056	0.066
4	405.1	5.1	9.0	0.054	0.056
6	429.7	8.3	9.1	0.047	0.050
8	390.5	8.7	9.1	0.043	0.054

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 $N\_DUP = 1$ or $N\_DUP = 4$ compared to using a single MPI process per node. When running multiple MPI processes per node, using $N\_DUP = 4$ is always faster than using $N\_DUP = 1$. It is surprising that, for the optimized SymmSquareCube algorithm, using $N\_DUP = 4$ with only 2 MPI processes per node is almost always faster than using $N\_DUP = 1$ 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, is 91.2% faster than the baseline performance without use of communication overlap.

For theoretical analysis, we estimate the inter-node communication volume, bandwidth, and time for the baseline SymmSquareCube algorithm using the multiple PPN overlap technique. We assume that all inter-process communication within a node uses shared memory and we report the inter-process communication volume that results from communication between processes on different nodes. We use the micro-benchmark in Section V-B to estimate the achievable bandwidth of broadcast and reduction operations when using different numbers of processes per node.

Table IV shows the estimated inter-node communication volume, bandwidth, and time as well as the actual inter-node communication time of the baseline SymmSquareCube algorithm. We observe that the inter-node communication volume increases when the number of MPI processes per node increases. This fact can discourage HPC developers from using multiple PPN to overlap communications. However, using more MPI processes per node allows collective operations to achieve a larger inter-node communication bandwidth, which reduces the time for inter-node communication.

E. SymmSquareCube using 2.5D Matrix Multiplication

We also test SymmSquareCube implemented using 2.5D matrix multiplication. This version of SymmSquareCube is shown in Algorithm 6. Our implementation of 2.5D matrix multiplication is based on that of Solomonik and Demmel [9] and uses a step of Cannon's algorithm as a subroutine. For simplicity, we skip the details of Cannon's algorithm, which includes point-to-point communications and local matrix multiplications. To optimize this version of SymmSquareCube by overlapping communications, each collective operation (steps 1, 3, and 5) is overlapped with itself using the nonblocking overlap technique. The algorithm does not present an opportunity to pipeline the communications like we did in Algorithm 5. We also test the implementation using multiple PPN overlap.

Algorithm 6 SymmSquareCube using 2.5D Matrix Multiplication

Input: matrix D distributed on $P_{:,:,1}$, communicators *row_comm*, *col_comm*, *grd_comm*, total number of processes p, and replication factor c

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

1: $P_{i,j,1}$ broadcasts $D_{i,j}$ as $A_{i,j}$ and $B_{i,j}$ to $P_{i,j,:}$ using *MPI_Bcast* with *grd_comm*
2: Perform $\sqrt{p/c^3}$ steps of Cannon's algorithm including local matrix multiplication and circular shift of A and B blocks using point-to-point communications in *row_comm* and *col_comm*
3: Allreduce sum $C_{i,j,k}$ to $D^2_{i,j}$ using *MPI_Allreduce* with *grd_comm*, $D^2_{i,j}$ will be used as the new $B_{i,j}$ in next step
4: Perform $\sqrt{p/c^3}$ steps of Cannon's algorithm including local matrix multiplication and circular shift of A and B blocks using point-to-point communications in *row_comm* and *col_comm*
5: Reduce sum $C_{i,j,k}$ to $D^3_{i,j}$ on $P_{i,j,1}$using *MPI_Reduce* with *grd_comm*

Table V shows the performance of the optimized 2.5D matrix multiplication version of SymmSquareCube using several process mesh configurations and replication factors c. The process mesh configurations are limited; $\sqrt{P/c}$ must be integral, where P is the total number of processes. Results are shown for $N\_DUP = 1$ and $N\_DUP = 4$ and for different numbers of PPN. The test molecular system is 1hsg_70.

The results show that the nonblocking overlap technique with $N\_DUP = 4$ consistently speeds up the non-overlapped case with $N\_DUP = 1$. However, the speedup is small, likely because *different* communication operations could not be overlapped and pipelined like in the 3D case. The results also show

TABLE V
PERFORMANCE OF THE OPTIMIZED 2.5D ALGORITHM VERSION OF SYMMSQUARECUBE WITH $N\_DUP = 1$ AND 4 FOR DIFFERENT NUMBERS OF PPN AND DUPLICATION FACTOR c.

2.5D Process Configuration			*Performance (TFlops)*	
PPN	$\sqrt{P/c} \times \sqrt{P/c} \times c$	*Total Nodes*	$N\_DUP = 1$	$N\_DUP = 4$
2	$8 \times 8 \times 2$	64	24.39	24.55
5	$12 \times 12 \times 2$	58	26.36	28.04
8	$16 \times 16 \times 2$	64	32.16	34.69
4	$9 \times 9 \times 3$	61	22.86	23.53
7	$12 \times 12 \times 3$	62	28.21	30.15
1	$4 \times 4 \times 4$	64	10.75	11.86
4	$8 \times 8 \times 4$	64	22.05	23.03
2	$5 \times 5 \times 5$	63	11.25	12.22
4	$6 \times 6 \times 6$	54	18.12	19.14
6	$7 \times 7 \times 7$	58	18.96	20.05
8	$8 \times 8 \times 8$	64	20.28	21.70

that, for the same replication factor c and thus same memory utilization, increasing *PPN* roughly improves performance.

VI. CONCLUSIONS

Overlapping communication operations, especially of collective operations, does not appear to be well-known, but the concept is simple and general and has the potential to be applied to a variety of applications. A new technique, called nonblocking overlap, can be used to incorporate communication overlapping into an existing code unobtrusively. Another technique is to use multiple PPN to overlap communications. Using this technique is nonintuitive, since using multiple PPN usually increases communication volume in distributed memory codes. However, this form of overlapping can also help utilize the available network bandwidth. To effectively use this multiple PPN overlap technique, we advocated using a different number of PPN for different stages of a code.

We have also developed a new implementation of a dense symmetric matrix squaring and cubing kernel that is useful for density matrix purification in electronic structure calculations. Our optimizations to SymmSquareCube in GTFock have been incorporated into the open-source software released at https://github.com/gtfock-chem. Although we treated the dense matrix case, the same ideas could be applied to the sparse matrix case.

In future work, we plan to investigate the effectiveness of overlapping communication operations in algorithms besides those of dense matrix computations. In distributed particle simulations, the forces between a set of particles can be arranged in a matrix that is partitioned using a 2D partitioning. This leads to algorithms that use collective communication along processor rows and columns of a processor mesh [17]. We also plan to investigate the use of overlapping communications in block iterative linear solvers, where reductions (vector norms and dot products) involving large numbers of nodes are the bottleneck.

ACKNOWLEDGMENTS

The authors gratefully acknowledge funding from an Intel Parallel Computing Center grant and funding from the National Science Foundation, grant ACI-1147843. The also authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

REFERENCES

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

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

[3] A. H. R. Palser and D. E. Manolopoulos, "Canonical purification of the density matrix in electronic-structure theory," *Physical Review B*, vol. 58, no. 19, pp. 12 704–12 711, 1998.

[4] D. R. Bowler and T. Miyazaki, "O(n) methods in electronic structure calculations," *Reports on Progress in Physics*, vol. 75, no. 3, p. 036503, 2012.

[5] 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. 104103, 2015.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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. ACM, 2015, pp. 4:1–4:8.

[13] 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.

[14] 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.

[15] Intel, "Intel MPI library for Linux OS developer reference," 2018. [Online]. Available: https://software.intel.com/sites/default/files/intelmpi-2019-developer-reference-linux.pdf

[16] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[17] S. Plimpton, "Fast parallel algorithms for short-range molecular-dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.