An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel® Xeon Phi^{TM} processor

Vladimir Mironov
Lomonosov Moscow State University
Leninskie Gory 1/3
Moscow 119991, Russian Federation
vmironov@lcc.chem.msu.ru

Yuri Alexeev
Argonne National Laboratory,
Leadership Computing Facility
Argonne, Illinois 60439, USA
yuri@alcf.anl.gov

Kristopher Keipert
Department of Chemistry and Ames
Laboratory, Iowa State University
Ames, Iowa 50011-3111, USA
kris@si.msg.chem.iastate.edu

Michael D'mello
Intel Corporation
Schaumburg, Illinois 60173, USA
michael.dmello@intel.com

Alexander Moskovsky
RSC Technologies
Moscow, Russian Federation
moskov@rsc-tech.ru

Mark S. Gordon
Department of Chemistry and Ames
Laboratory, Iowa State University
Ames, Iowa 50011-3111, USA
mgordon@iastate.edu

ABSTRACT

Modern OpenMP threading techniques are used to convert the MPI-only Hartree-Fock code in the GAMESS program to a hybrid MPI/OpenMP algorithm. Two separate implementations that differ by the sharing or replication of key data structures among threads are considered, density and Fock matrices. All implementations are benchmarked on a super-computer of 3,000 Intel® Xeon Phi^{TM} processors. With 64 cores per processor, scaling numbers are reported on up to 192,000 cores. The hybrid MPI/OpenMP implementation reduces the memory footprint by approximately 200 times compared to the legacy code. The MPI/OpenMP code was shown to run up to six times faster than the original for a range of molecular system sizes.

CCS CONCEPTS

•Theory of computation → Massively parallel algorithms; •Computing methodologies → Quantum mechanic simulation; Massively parallel and high-performance simulations;

KEYWORDS

Quantum chemistry, parallel Hartree-Fock, parallel Self Consistent Field, OpenMP, MPI, GAMESS

ACM Reference format:
Vladimir Mironov, Yuri Alexeev, Kristopher Keipert, Michael D'mello, Alexander Moskovsky, and Mark S. Gordon. 2017. An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel® Xeon Phi^{TM} processor. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.
DOI: 10.1145/3126908.3126956

1 INTRODUCTION

The field of computational chemistry encompasses a wide range of empirical, semi-empirical, and *ab initio* methods that are used to compute the structure and properties of molecular systems. These methods therefore have a significant impact on not only chemistry, but materials, physics, engineering and the biological sciences as well. *Ab initio* methods are rigorously derived from quantum mechanics. In principle, *ab initio* methods are more accurate than methods with empirically fitted parameters. Unfortunately, this accuracy comes at significant computational expense. For example, the time to solution for Hartree-Fock (HF) and Density Functional Theory (DFT) methods scale as approximately $O(N^3)$, where N is the number of degrees of freedom in the molecular system. The HF solution is commonly used as a starting point for more accurate *ab initio* methods, such as second order perturbation theory and coupled-cluster theory with single, double, and perturbative triple excitations. These post-HF methods scale as $O(N^5)$ and $O(N^7)$, respectively. These computational demands clearly require efficient utilization of parallel computers to treat increasingly large molecular systems with high accuracy.

Modern high performance computing hardware architecture has substantially changed over the last 10 to 15 years. Nowadays, a "many-core" philosophy is common to most platforms. For example, the Intel Xeon Phi processor can have up to 72 cores. For good resource utilization, this necessitates (hybrid) MPI+X parallelism in application software.

The subject of this work is the successful adaptation of the HF method in the General Atomic and Molecular Electronic Structure System (GAMESS) quantum chemistry package to the second-generation Intel Xeon Phi processor platform. GAMESS is a free quantum chemistry software package maintained by the Gordon research group at Iowa State University [15]. GAMESS has been cited more than 10,000 times in the literature, downloaded more than 30,000 times and includes a wide array of quantum chemistry methods. The objective here is to start with the MPI-only version of GAMESS HF and systematically introduce optimizations which improve performance and reduce the memory footprint. Many existing methods in GAMESS are parallelized with MPI. OpenMP is an attractive high-level threading application program interface (API) that is scalable and portable. The OpenMP interface conveniently enables sharing of the two major objects in the HF self-consistent field (SCF) loop: the density matrix and the Fock matrix.

The density and Fock data structures account for the majority of the memory footprint of each MPI process. Indeed, since these two objects are replicated across the MPI processes, memory capacity limits can easily come into play if one tries to improve the time to solution using a large number of cores. By sharing one or both of the aforementioned objects between threads, one can reduce the memory footprint and more easily leverage all of the resources (cores, fast memory etc.) of the Intel Xeon Phi processor. Reducing the memory footprint is also expected to lead to better cache utilization, and, therefore, enhanced performance. Two hybrid OpenMP/MPI implementations of the publicly available version of the GAMESS (MPI-only) code base were constructed for this work. The first version is referred to as the "shared density private Fock", or "private Fock" version of the code. The second version is referred to as the "shared density shared Fock", or "shared Fock" version.

In the following section, a brief survey of related work is presented. Next, key algorithmic features of the HF self-consistent field (SCF) method are discussed. Then, a description of the computer hardware test bed that was used for benchmarking purposes is presented. An explanation of the code transformations employed in the hybrid implementation in this work follows. Next, the memory and time-to-solution results of the hybrid approach are shown. Results on up to 3,000 Intel Xeon Phi processors are presented for a range of chemical system sizes. The work ends with concluding remarks and a discussion of directions for future work.

2 RELATED WORK

The HF algorithm has been a primary parallelization target since the onset of parallel computing. The primary computational components of the HF algorithm are construction of the density and Fock matrices, that are described in section 3 of this work. The irregular task and data access patterns during Fock matrix construction bring significant challenges to efficient parallel distribution of the computation. The poor scaling of Fock matrix diagonalization is a major expense as well. Linear scaling methods like the fragment molecular orbital method (FMO) have been successfully applied to thousands of atoms and CPU cores [5, 27], but such methods introduce additional approximations [11, 12]. In any case, fragmentations methods may benefit from optimizations of the core HF algorithm as well.

Early HF parallelization efforts focused on the distributed computation of the many electron repulsion integrals (ERIs) required for Fock matrix construction via MPI or other message passing libraries. The Fock and density matrices were often replicated for each rank, and load balancing algorithms were a primary optimization target. Blocking and clustering techniques were explored in depth in a landmark paper by Foster et al. [14]. Their contributions were implemented in the quantum chemistry package NWChem [28]. In a follow-up paper by Harrison et al. [16], a node-distributed HF implementation was introduced. In this work, both the density and Fock matrices were distributed across nodes using globally addressable array (GA). In a more recent work UPC++ library was used to achieve this goal [22]. A similar approach was used to implement distributed data parallel HF by [4, 6] in the GAMESS code. This implementation utilizes the Distributed Data Interface (DDI) message passing library [13]. To further address the load balancing issues, a work stealing technique was introduced by Liu et al. [20].

A detailed study and analysis of the scalability of Fock matrix construction and density matrix construction [10], including the effects of load imbalance, was explored in a work by Chow et al. [9]. In this work, density matrix construction was achieved by density purification techniques and the resulting implementation was scaled up to 8,100 Tianhe-2 Intel Xeon Phi first generation co-processors. In fact, a number of attempts have been made to design efficient implementations of HF for accelerators [8, 9, 25, 26, 29] and other post-HF methods [7]. A major issue in this context is the management of shared data structures between cores – in particular, the density and Fock matrices. OpenMP HF implementations with a replicated Fock matrix and shared density matrix have been explored in the work of Ishimura et al. [17] and Mironov et al. [21]. The differences between these works are in the workload distribution among MPI ranks and OpenMP threads. The current work borrows some techniques from these previous works which implement HF for accelerators. The result is a hybrid MPI/OpenMP implementation that is designed to scale well on a large number of Intel Xeon Phi processors, while at the same time managing the memory footprint and maintaining compatibility with the original GAMESS codebase.

3 HARTREE-FOCK METHOD

The HF method is used to iteratively solve the electronic Schrödinger equation for a many-body system. The resulting electronic energy and electronic wave function can be used to compute equilibrium geometries and a variety of molecular properties. The wave function is constructed of a finite set of basis functions suitable for algebraic representation of the integro-differential HF equations. Central to HF is an effective one-electron Hamiltonian called the Fock operator which describes electron-electron interactions by mean field theory. In computational practice, the Fock operator is defined in matrix form (Fock matrix). The HF working equations are then represented by a nonlinear eigenvalue problem called the Hartree-Fock equations:

$$\mathbf{FC} = \epsilon \mathbf{SC} \tag{1}$$

where ϵ is a diagonal matrix corresponding to the electronic orbital energies, $\mathbf{F}$ is a Fock matrix, $\mathbf{C}$ is matrix of molecular orbital (MO) coefficients, and $\mathbf{S}$ is the overlap matrix of the atomic orbital (AO) basis set. The HF equations are solved numerically by self-consistent field (SCF) iterations.

The SCF iterations are preceded by computation of an initial guess density matrix and core Hamiltonian. An initial Fock matrix is constructed from terms of the core Hamiltonian and a symmetric orthogonalization matrix. Next, the Fock matrix is diagonalized to provide the MO coefficients $\mathbf{C}$. These MO coefficients are used to compute an initial guess density matrix. The SCF iterations follow, in which a new Fock matrix is constructed as a function of the

Algorithm 1 MPI parallelization of SCF in stock GAMESS

```
 1: for $i = 1, NShells$ do
 2:     for $j = 1, i$ do
 3:         call ddi_dlbnext($ij$)    ▷ MPI DLB: check I and J indices
 4:         for $k = 1, i$ do
 5:             $k{==}i\ ?\ l_{max} \leftarrow k : l_{max} \leftarrow j$
 6:             for $l = 1, l_{max}$ do
    ▷ Schwartz screening:
 7:                 $screened \leftarrow$ schwartz($i, j, k, l$)
 8:                 if not $screened$ then
 9:                     call eri($i, j, k, l, X_{ijkl}$)    ▷ Calculate $(i, j|k, l)$
    ▷ Update process-local 2e-Fock matrix:
10:                     $Fock_{ij,kl,ik,jl,il,jk}$ +=
                            $X_{ijkl} \cdot D_{kl,ij,jl,ik,jk,il}$
11:                 end if
12:             end for
13:         end for
14:     end for
15: end for
    ▷ 2e-Fock matrix reduction over MPI ranks:
16: call ddi_gsumf($Fock$)
```

guess density matrix. Diagonalization of the updated Fock matrix provides a new set of MO coefficients which are used to update the density matrix. This iterative process continues until convergence is reached, which is defined by the root-mean-squared difference of consecutive densities lying below a chosen convergence threshold.

Contrary to what one might expect, the most time-consuming part of the calculation is not the solution of the Hartree-Fock equations, but rather the construction of the Fock matrix [18]. The calculation of the Fock matrix elements can be separated into one-electron and two-electron components. The computational complexity of these two parts is $O(N^2)$ and $O(N^4)$, respectively. In most cases of practical interest, the calculation of the two-electron contribution to the Fock matrix occupies the majority of the overall compute time.

4 OPTIMIZATION AND PARALLELIZATION OF THE HARTREE-FOCK METHOD

4.1 General considerations and design

In this section, three implementations of the HF algorithm are presented: the original MPI algorithm [24] and two new hybrid MPI/OpenMP algorithms. As mentioned earlier, the most expensive steps in HF are the computation of ERIs and the contribution of ERIs multiplied by corresponding density elements during construction of the Fock matrix. The symmetry-unique ERIs are labeled in four dimensions over i, j, k, l shell[1] indices. The symmetry-unique quartet shell indices are traversed during Fock matrix construction. Parallelization over the four indices is complicated by the high order of permutational symmetry for shell indices. In addition, many integrals are very small in magnitude and are screened out using the Cauchy-Schwarz inequality equation (see e.g. [18, p. 118]). Each ERI is used to construct six elements of the Fock matrix shown in eqs. (2a)–(2f) where $(i, j|k, l)$ corresponds to a single ERI:

$$F_{ij} \leftarrow (i, j|k, l) \cdot D_{kl}; \tag{2a}$$
$$F_{kl} \leftarrow (i, j|k, l) \cdot D_{ij}; \tag{2b}$$
$$F_{ik} \leftarrow (i, j|k, l) \cdot D_{jl}; \tag{2c}$$
$$F_{jl} \leftarrow (i, j|k, l) \cdot D_{ik}; \tag{2d}$$
$$F_{il} \leftarrow (i, j|k, l) \cdot D_{jk}; \tag{2e}$$
$$F_{jk} \leftarrow (i, j|k, l) \cdot D_{il}; \tag{2f}$$

The irregular storage and access of ERIs during Fock matrix construction is a significant computational challenge. Also, the Fock matrix construction is distributed among ranks, and the final Fock matrix is summed up by a reduction. A detailed explanation of the SCF implementation in GAMESS can be found elsewhere [24].

4.2 MPI-based Hartree-Fock algorithm

The MPI parallelization in the official release of the GAMESS code is shown in Algorithm 1. While this implementation has been remarkably successful, it has the disadvantage of a very high memory footprint. This is because a number of data structures (including the density matrix, the atomic orbital overlap matrix, and the one- and two-electron contributions to the Fock matrix) are replicated across MPI ranks. It is a major issue for processors which have a large number of cores (like the Intel Xeon Phi). For example, running 256 MPI ranks on a single Intel Xeon Phi processor increases the memory footprint for both density and Fock matrices by a factor of 256 times. This implementation is therefore severely restricted when it comes to the size of the chemical systems that can be made to fit in memory.

In a typical calculation, the number of shells (see *NShells* in Algorithm 1) is less than one thousand. Most often, the number can be on the order of a few hundred shells. Thus, parallelization over a two shell indices (Algorithm 1) frequently results in load imbalances. The HF algorithm in GAMESS was originally designed for small- to medium-sized x86 CPU architecture clusters when load balancing is not such a significant issue. However, switching to computer systems with larger parallelism (large number of compute nodes) requires a change of approach for load balancing. Multiple solutions exist for this problem. Perhaps the simplest one is to use more shell indices to increase the iteration space and improve the load balance or introduce multilevel load balancing schemes.

4.3 Hybrid OpenMP/MPI Hartree-Fock algorithm

In this section, the hybrid MPI/OpenMP two-electron Fock matrix code implementations of the current work are described. The main goal of this implementation is to reduce the memory footprint of the MPI-based code and to improve the load balancing by utilizing the OpenMP runtime library.

Modern computational cluster nodes can have a large number of cores operating on a single random access memory. In order to efficiently utilize all of the available CPU cores, it is necessary to run many threads of execution. The major disadvantage of an MPI-only HF code is that all of the data structures are replicated

[1]By term *shell* we mean a group of basis set functions related to the same atom and sharing same set of internal parameters. Grouping basis functions into shells is a common technique in Gaussian-based quantum chemistry codes like GAMESS.

Algorithm 2 Hybrid MPI-OpenMP SCF algorithm; Fock matrix is replicated across all threads i.e. Fock matrix is private.

```
 1: !$omp parallel private($j, k, l, l_{max}, X_{ijkl}$) shared($I$)
        reduction(+ : $Fock$)
 2: loop
 3:     !$omp master
 4:     call ddi_dlbnext($i$)                ▷ MPI DLB: get new I index
 5:     !$omp end master
 6:     !$omp barrier
 7:     !$omp do collapse(2) schedule(dynamic,1)
 8:     for $j$ = 1, $i$ do
 9:         for $k$ = 1, $i$ do
10:             $k{=}{=}i \; ? \; l_{max} \leftarrow k : l_{max} \leftarrow j$
11:             for $l$ = 1, $l_{max}$ do
    ▷ Schwartz screening:
12:                 $screened \leftarrow$ schwartz($i, j, k, l$)
13:                 if not $screened$ then
14:                     call eri($i, j, k, l, X_{ijkl}$)   ▷ Calculate $(i, j|k, l)$
    ▷ Update private 2e-Fock matrix:
15:                     $Fock_{ij,kl,ik,jl,il,jk}$ +=
                            $X_{ijkl} \cdot D_{kl,ij,jl,ik,jk,il}$
16:                 end if
17:             end for
18:         end for
19:     end for
20:     !$omp end do
21: end loop
22: !$omp end parallel
23: call ddi_gsumf($Fock$)       ▷ 2e-Fock matrix reduction over MPI
```

across MPI processes (ranks) – since to spawn a process is the only way to use a CPU core. In practice, it is found that the memory footprint gets prohibitive rather quickly as the chemical system is scaled up. It follows from Algorithm 1 that only the Fock matrix update incurs a potential race-condition (write dependencies) when leveraging multiple threads. Other large memory objects like the density matrix, the atomic orbital overlap matrix, and others do not exhibit this problem, because they are read-only matrices, and as a result they can be safely shared across all threads for each MPI rank.

In a first attempt, a hybrid MPI/OpenMP Hartree-Fock code was developed with the Fock matrix replicated across threads (Algorithm 2). This is what is referred to as the private Fock (hybrid) version of the code. In the first loop, the master thread of each MPI rank updates the i index. This operation is protected by implicit and explicit barriers. OpenMP parallelization is implemented over combined j and k shell loops. Joining loops provides a much larger pool of tasks and thereby alleviates any load balancing issues that may arise. To lend credence to this idea, static and dynamic schedules of OpenMP were tested for the collapsed loop. No significant difference between the various OpenMP load balancer modes was observed. The l loop is the same as in the original implementation of GAMESS. The last step is the same as in the MPI-based algorithm: reduction of the Fock matrix over MPI processes.

Sharing all of the large matrices except the Fock matrix saves an enormous amount of memory on the multicore systems. The observed memory footprints on the latest Xeon and Xeon Phi CPUs were reduced about 5 times. However, the ultimate goal of this work is to move all of the large data structures to shared memory.

It is not straightforward to remove Fock matrix write dependencies in the OpenMP region. As shown in eqs. (2a)–(2f), up to six Fock matrix elements are updated at one time by each thread. The ERI contribution is added to the three shell column-blocks of the Fock matrix simultaneously – namely the i, j, and k blocks. Each block corresponds to one shell and to all basis set functions associated with this shell. The main idea of the present approach is to use thread-private storage for each of these blocks. They are used as a buffer accumulating partial Fock matrix contribution and help to avoid write dependency. Partial Fock matrix contributions are flushed to the full matrices when the corresponding shell index changes.

The access pattern of the Fock matrix by k index corresponds to only one Fock matrix element. If threads have different k and l shell indices, it would be possible to skip saving data to the k buffer and instead, to directly update the corresponding parts of the full Fock matrix. This condition will be satisfied if OpenMP parallelization over k and l loops is used. In this case, private storage is necessary for only the i and j blocks of the Fock matrix.

In the shared Fock matrix algorithm (Algorithm 3) the original four loops (Algorithm 1) are arranged into two merged index loops. The first and second loops correspond to the combined ij and kl indices, respectively. MPI parallelization is executed over the top (ij) loop, while OpenMP parallelization is accomplished over the inner (kl) loop. In contrast to the private Fock matrix algorithm (Algorithm 2), this partitioning favors computer systems with a large number of MPI ranks and is the preferred strategy because this implementation of MPI iteration space is larger and the load balance is finer. By using this partitioning, it is also possible to utilize Schwarz screening across the i and j indices. Partitioning is especially important for very large jobs with very sparse ERI tensor because it allows the user to completely skip the most costly top-loop iterations.

Another difference from the private Fock matrix algorithm is that the ERI contribution is now added in three places (Algorithm 3, lines 25-27): to the private i buffer (F_{ij}, F_{ik}, F_{il}), the private j buffer (F_{jk}, F_{jl}), and the shared Fock matrix (F_{kl}). At the end of the joint kl-loop, the partial Fock matrix contribution from i and j buffers needs to be added to the full Fock matrix. It is computationally expensive for a multithreaded environment because it requires explicit thread synchronization. However, it is possible to reduce the frequency of i buffer flushing. After each kl loop, the i index very likely remains the same and there will be no need for i buffer flushing. In the present algorithm, the old i index is saved after the kl loop (Algorithm 3, line 33). The flushing of the i buffer contribution to the Fock matrix is only done if the i index were changed since the last iteration. Flushing the j buffer is still required after each kl loop (Algorithm 3, line 31).

A special array structure is needed for flushing and reducing buffers for the i and j blocks. Buffers are organized as two-dimensional arrays. The outer dimension of these arrays corresponds to threads,

Algorithm 3 Hybrid MPI-OpenMP SCF algorithm; Fock matrix is shared across all threads.

```
 1: $mxsize \leftarrow$ ubound($Fock$)·$shellSize$
 2: $nthreads \leftarrow$ omp_get_max_threads()
 3: allocate($F_I(mxsize, nthreads, F_J(mxsize, nthreads)$)
 4: !$omp parallel shared($F_I, F_J, Fock$) &
        private($i, j, k, l, ithread$)
 5: $ithread \leftarrow$ omp_get_thread_num()
 6: loop
 7:     !$omp master
 8:     call ddi_dlbnext($ij$) ▷ MPI DLB: get new combined IJ index
 9:     !$omp end master
10:     !$omp barrier
11:     $i, j \leftarrow ij$                          ▷ Deduce I and J indices
12:     $kl_{max} \leftarrow i, j$                      ▷ Deduce KL-loop limit
13:     $screened \leftarrow$ schwartz($i, j, i, j$)        ▷ I and J prescreening
14:     if not $screened$ then
15:         if $i \neq i_{old}$ then              ▷ If $i$ was changed flush $F_I$
16:             $Fock(:, i)$+=$\sum F_I(:, 1{:}nthreads)$
17:             !$omp barrier
18:         end if
19:         !$omp do schedule(dynamic,1)
20:         for $kl = 1, kl_{max}$ do
21:             $k, l \leftarrow kl$                   ▷ Deduce K and L indices
22:             $screened \leftarrow$ schwartz($i, j, k, l$) ▷ Schwartz screening
23:             if not $screened$ then
24:                 call eri($i, j, k, l, X_{ijkl}$)      ▷ Calculate $(i, j|k, l)$
    ▷ Update private partial Fock matrices:
25:                 $F_I(:, ithread)_{j,k,l}$+=$X_{ijkl} \cdot D_{kl,jl,jk}$
26:                 $F_J(:, ithread)_{k,l}$+=$X_{ijkl} \cdot D_{il,ik}$
    ▷ Update shared Fock matrix:
27:                 $Fock(k, l)$+=$X_{ijkl} \cdot D(i, j)$
28:             end if
29:         end for
30:         !$omp end do
31:         $Fock(:, j)$+=$\sum F_J(:, 1{:}nthreads)$           ▷ Flush $F_J$
32:         !$omp barrier
33:         $i_{old} \leftarrow i$
34:     end if
35: end loop
    ▷ Flush remainder $F_i$ contribution to $Fock$:
36: $Fock(:, i)$+=$\sum F_I(:, 1{:}nthreads)$
37: !$omp end parallel
38: call ddi_gsumf($Fock$)    ▷ 2e-Fock matrix reduction over MPI
```

and the inner dimension corresponds to the data. Using Fortran notation, data is stored in matrix columns, with each thread displayed in its own column. This (column-wise) access pattern is used when threads add an ERI contribution to the buffers (Figure 1 (A)). The access pattern is different when it is necessary to flush a buffer into the full Fock matrix. The tree-reduction algorithm is used to sum up the contribution from different columns and add them to the full Fock matrix. In this case, the access of threads to this matrix is row-wise (Figure 1 (B)). Padding bytes were added to the leading dimension of the array and chunking was used on the reduction step to prevent false sharing. After the buffer is flushed into the

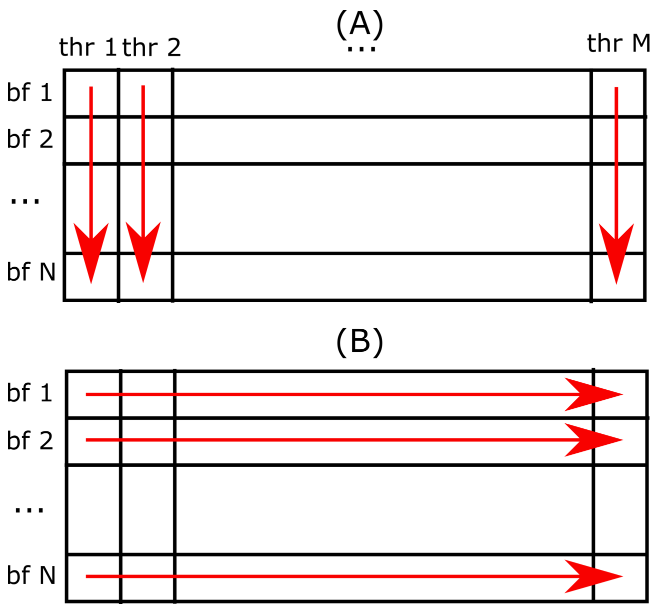

Figure 1: I and J Fock vectors update (A) and summing up all Fock elements for each Fock element in the vectors (B). "bf" means basis function and "thr" means thread.

Fock matrix, it is filled in with zeroes and is ready for the next cycle.

5 METHODOLOGY

5.1 Description of hardware and software

The benchmarks reported in this paper were performed on the Intel Xeon Phi systems provided by the Joint Laboratory for System Evaluation (JLSE) and the Theta supercomputer at the Argonne Leadership Computing Facility (ALCF) [1], which is a part of the U.S. Department of Energy (DOE) Office of Science (SC) Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program [3]. Theta is a 10-petaflop Cray XC40 supercomputer consisting of 3,624 Intel Xeon Phi 7230 processors. Hardware details for the JLSE and Theta system are shown in Table 1.

The Intel Xeon Phi processor used in this paper has 64 cores each equipped with L1 cache. Each core also has two Vector Processing Units, both of which need to be used to get peak performance. This is possible because the core can execute two instructions per cycle. In practical terms, this can be achieved by using two threads per core. Pairs of cores constitute a tile. Each tile has an L2 cache symmetrically shared by the core pair. The L2 caches between tiles are connected by a two dimensional mesh. The cores themselves operate at 1.3 GHz. Beyond the L1 and L2 cache structure, all the cores in the Intel Xeon Phi processor share 16 GBytes of MCDRAM (also known as High Bandwidth Memory) and 192 GBytes of DDR4. The bandwidth of MCDRAM is approximately 400 GBytes/sec while the bandwidth of DDR4 is approximately 100 GBytes/sec.

These two levels of memory can be configured in three different ways (or modes). The modes are referred to as Flat mode, Cache mode, and Hybrid mode. Flat mode treats the two levels of memory

Table 1: Hardware and software specifications

Intel Xeon Phi node characteristics	
Intel Xeon Phi models	7210 and 7230 (64 cores, 1.3 GHz, 2,622 GFLOPs)
Memory per node	16 GB MCDRAM, 192 GB DDR4 RAM
Compiler	Intel Parallel Studio XE 2016v3
JLSE Xeon Phi cluster (26.2 TFLOPS peak)	
# of Intel Xeon Phi nodes	10
Interconnect type	Intel Omni-Path^{TM}
Theta supercomputer (9.65 PFLOPS peak)	
# of Intel Xeon Phi nodes	3,624
Interconnect type	Aries interconnect with Dragonfly topology

Table 2: Chemical systems used in benchmarks and their size characteristics

Name	# atoms	# BFs[a]	Memory footprint[b], GB		
			MPI[c]	Pr.F.[d]	Sh.F.[e]
0.5 nm	44	660	7	0.13	0.03
1.0 nm	120	1800	48	1	0.2
1.5 nm	220	3300	160	3	0.8
2.0 nm	356	5340	417	8	2
5.0 nm	2016	30 240	9869	257	52

[a] BF – basis function [b] Estimated using eqs. (3a)–(3c)
[c] MPI-only SCF code [d] Private Fock SCF code
[e] Shared Fock SCF code

as separate entities. The Cache mode treats the MCDRAM as a direct mapped L3 cache to the DDR4 layer. Hybrid mode allows the user to use a fraction of MCDRM as L3 cache allocate the rest of the MCDRAM as part of the DDR4 memory. In Flat mode, one may choose to run entirely in MCDRAM or entirely in DDR4. The ”numactl” utility provides an easy mechanism to select which memory is used. It is also possible to choose the kind of memory used via the ”memkind” API, though as expected this requires changes to the source code.

Beyond memory modes, the Intel Xeon Phi processor supports five cluster modes. The motivation for these modes can be understood in the following manner: to maintain cache coherency the Intel Xeon Phi processor employs a distributed tag directory (DTD). This is organized as a set of per-tile tag directories (TDs), which identify the state and the location on the chip of any cache line. For any memory address, the hardware can identify the TD responsible for that address. The most extreme case of a cache miss requires retrieving data from main memory (via a memory controller). It is therefore of interest to have the TD as close as possible to the memory controller. This leads to a concept of locality of the TD and the memory controllers. It is in the developer’s interest to maintain the locality of these messages to achieve the lowest latency and greatest bandwidth of communication with caches. Intel Xeon Phi supports all-to-all, quadrant/hemisphere and sub-NUMA cluster SNC-4/SNC-2 modes of cache operation.

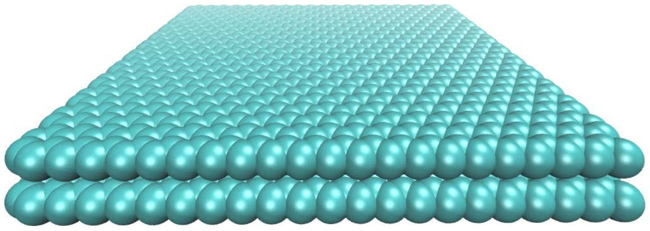
Figure 2: Model system of a C_{2016} graphene bilayer. In the text, we refer to this system as 5 nm. There are two layers with size 5 nm by 5 nm. Each graphene layer consists of 1,008 carbon atoms.

For large problem sizes, different memory and clustering modes were observed to have little impact on the time to solution for the three versions of the GAMESS code. For this reason, we simply chose the mode most easily available to us. In other words, since the choice of mode made little difference in performance, our choice of Quad-Cache mode was ultimately driven by convenience (this being the default choice in our particular environment). Our comments here apply to large problem sizes, so for small problem sizes, the user will have to experiment to find the most suitable mode(s).

5.2 Description of chemical systems

For benchmarks, a system consisting of parallel series of graphene sheets was chosen. This system is of interest to researchers in the area of (micro)lubricants [19]. A physical depiction of the configuration is provided in Figure 2. The graphene-sheet system is ideal for benchmarking, because the size of the system is easily manipulated. Various Fock matrix sizes can be targeted by adjusting the system size.

5.3 Characteristics of datasets

In all, five configurations of the graphene sheets system were studied. The datasets for the systems studied are labeled as follows: 0.5 nm, 1.0 nm, 1.5 nm, 2.0 nm, and 5.0 nm. Table 2 lists size characteristics of these configurations. The same 6-31G(d) basis set (per atom) was used in all calculations. For N basis functions, the density, Fock, AO overlap, one-electron Fock matrices and the matrix of MO coefficients are N×N in size. These are the main data structures of significant size. Therefore, the benchmarks performed in this work process matrices which range from 660×660 to 30,240×30,240. For each of the systems studied, Table 2 lists the memory requirements of the three versions of GAMESS HF code. Denoting N_{BF} as the number of basis functions, the following equations describe the asymptotic ($N_{BF} \to \infty$) memory footprint for the studied HF algorithms:

$$M_{MPI} = 5/2 \cdot N_{BF}^2 \cdot N_{MPI\_per\_node}, \tag{3a}$$

$$M_{PrF} = (2 + N_{threads}) \cdot N_{BF}^2 \cdot N_{MPI\_per\_node}, \tag{3b}$$

$$M_{ShF} = 7/2 \cdot N_{BF}^2 \cdot N_{MPI\_per\_node}, \tag{3c}$$

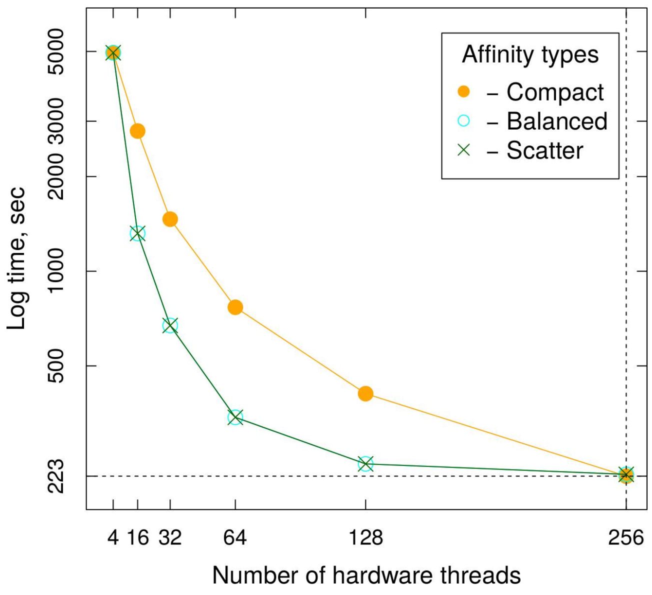

Figure 3: Performance dependence on OpenMP thread affinity type for the shared Fock version of the GAMESS code on a single Intel® Xeon Phi^{TM} processor using the 1.0 nm benchmark. All calculations are performed in quad-cache mode. Four MPI ranks were used in all cases. The number of threads per MPI rank was varied from 1 to 64.

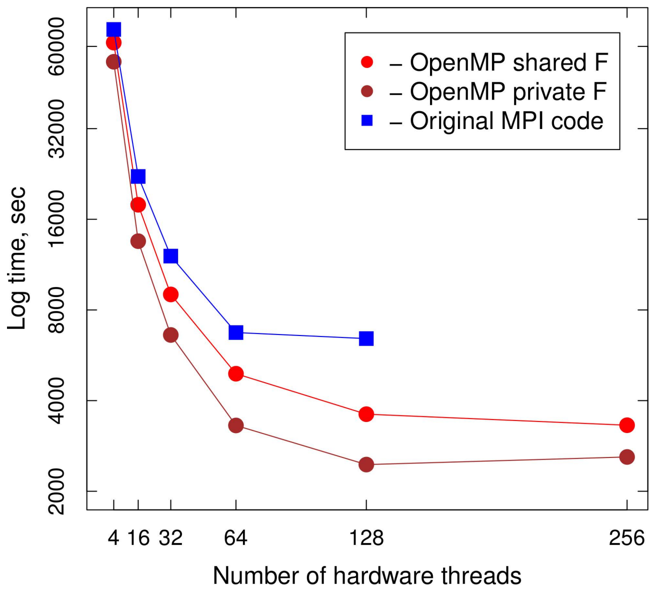

Figure 4: Scalability with respect to the number of hardware threads of the original MPI code and two OpenMP versions on a single Intel® Xeon Phi^{TM} processor using the 1.0 nm benchmark.

where M_{MPI}, M_{PrF}, M_{ShF} denote the memory footprint of MPI-only, private Fock, and shared Fock algorithms respectively; $N_{threads}$ denotes the number of threads per MPI process for the OpenMP code, and $N_{MPI\_per\_node}$ denotes the number of MPI processes per KNL node. For OpenMP runs $N_{MPI\_per\_node} = 4$, while for MPI runs the number of MPI ranks was varied from 64 to 256.

If one compares columns MPI versus Pr.F and Sh.F. in Table 2, you will see that the private Fock code has about a 50 times less footprint compared to the stock MPI code. For the shared Fock code, the difference is even more dramatic with a savings of about 200 times. The ideal difference is 256 times since we compare 256 MPI ranks in the stock MPI code where all data structures are replicated versus 1 MPI rank with 256 threads for the hybrid MPI/OpenMP codes. But we introduced additional replicated structures (see Figure 1) and many relatively small data structures are replicated also in the MPI/OpenMP codes. This explains the difference between the ideal and observed footprints.

Each of the aforementioned datasets was used to benchmark three versions of the GAMESS code. The first version is the stock GAMESS MPI-only release that is freely available on the GAMESS website [2]. The second version is a hybrid MPI/OpenMP code, derived from the stock release. This version has a shared density matrix, but a thread-private Fock matrix. The third version of the code is in turn derived from the second version; it has shared density and Fock matrices. A key objective was to see how these incremental changes allow one to manage (i.e., reduce) the memory footprint of the original code while simultaneously driving higher performance.

6 RESULTS

6.1 Single node performance

The second generation Intel Xeon Phi processor supports four hardware threads per physical core. Generally, more threads per core can help hide latencies inherent in an application. For example, when one thread is waiting for memory, another can use the processor. The out-of-order execution engine is beneficial in this regard as well. To manipulate the placement of processes and threads, the `I_MPI_DOMAIN` and `KMP_AFFINITY` environment variables were used. We examined the performance picture when one thread per core is utilized and when four threads per core are utilized. As expected, the benefit is highest for all versions of GAMESS for two threads (or processes) per core. For three and four threads per core, some gain is observed, albeit at a diminished level. Figure 3 shows the scaling curves with respect to the number of hardware threads utilized observed by us.

As a first test, single-node scalability was examined with respect to hardware threads of all three versions of GAMESS. For the MPI-only version of GAMESS, the number of ranks was varied from 4 to 256. For the hybrid versions of GAMESS, the number of ranks times the number of threads per rank is the number of hardware threads targeted. The larger memory requirements of the original MPI-only code restrict the computations to, at most, 128 hardware threads. In contrast, the two hybrid versions can easily utilize all 256 hardware threads available. Finally, in general terms, on cache based memory architectures, it is expected that larger memory footprints potentially lead to more cache capacity and cache line conflict effects. These effects can lead to diminished performance, and this is yet another motivation to look at a hybrid MPI+X approach.

The results of our single-node tests are plotted in Figure 4. It is found that using the private Fock version leads to the best time to solution for the 1.0 nm dataset, for any number of hardware threads. This version of the code is much more memory-efficient than the stock version but, because the Fock matrix data structure is private, it has a much larger memory footprint than the shared Fock version of GAMESS. Nevertheless, because the Fock matrix is private, there is less thread contention than the shared Fock version.

It was mentioned in Section 4.3 that shared Fock algorithm introduces additional overhead for thread synchronization. For small numbers of Intel Xeon Phi threads, this overhead is expected to be low. Therefore the shared Fock version is expected to be on par with the other versions. Eventually, as the overhead of the synchronization mechanisms begins to increase, the private Fock version of the code is found to dominate. In the end, the private Fock version outperforms stock GAMESS because of the reduced memory footprint, and outperforms the shared Fock version because of a lower synchronization overhead. Therefore, on a single node, the private Fock version gives the best time-to-solution of the three codes, but the shared Fock version strikes a (better) balance between memory utilization and performance.

Beyond this, one must consider the choice of memory mode and cluster mode of the Intel Xeon Phi processor. It should be noted that, depending on the compute and memory access patterns of a code, the choice of memory and cluster mode can be a potentially significant performance variable. The performance impact of different memory and cluster modes is examined for the 0.5 nm (small) and 2.0 nm (large) datasets. The results are shown in Figure 5. For both datasets, some variation in performance is apparent when different cluster modes and memory modes are used. The smaller dataset indicates more sensitivity to these variables than the larger dataset. Also, for both data sizes the private Fock version performs best in all cluster and memory modes tested. Also, except in the All-to-All cluster mode, the shared Fock version significantly outperforms the MPI-only stock version. In the All-to-All mode, the MPI-only version actually outperforms the shared Fock version for small datasets, and the two versions are close to parity for large datasets. In total, it is concluded that the quadrant-cache cluster-memory mode is best suited to the design of the GAMESS hybrid codes.

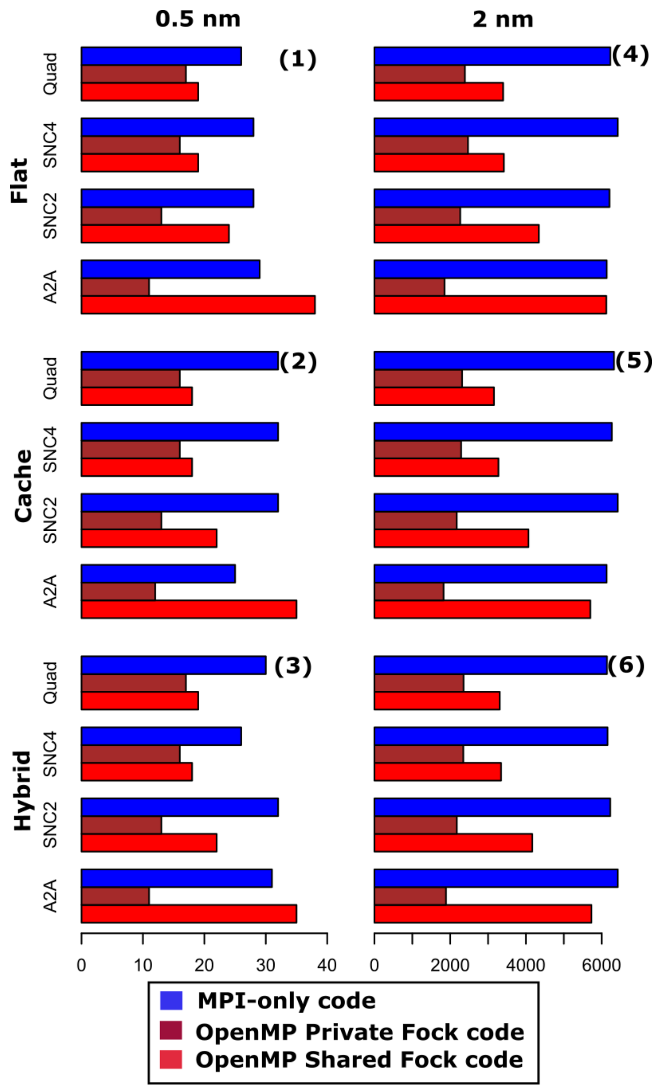

Figure 5: Time to solution (x axis, time in seconds) for different clustering and memory modes. Left column displays the small chemical system – 0.5 nm bilayer graphene and right column displays one of the largest molecules bilayer graphene – 2.0 nm.

6.2 Multi-node performance

It is very important to note that the total number of MPI ranks for GAMESS is actually twice the number of compute ranks because of the DDI. The DDI layer was originally implemented to support one-sided communication using MPI-1. For GAMESS developers, the benefit of DDI is convenience in programming. The downside is that each MPI compute process is complemented by an MPI data server (DDI) process, which clearly results in increased memory requirements. Because data structures are replicated on a rank-by-rank basis, the impact of DDI on memory requirements is particularly unfavorable to the original version of the GAMESS code. To alleviate some of the limitations of the original implementation, an implementation of DDI based on MPI-3 was developed [23]. Indeed, by leveraging the "native" support of one-sided communication in MPI-3, the need for a DDI process alongside each MPI rank was eliminated. For all three versions of the code benchmarked here, no DDI processes were needed.

Figure 6 shows the multi-node scalability of the MPI-only version of GAMESS versus the private Fock and the shared Fock hybrid versions. It is important to appreciate at the outset that the multi-node scalability of the original MPI-only version of GAMESS is already reasonable. For example, the code scales linearly to 256 Xeon Phi nodes, and it is really the memory footprint bottleneck that limits how well all the Xeon Phi cores on any given node can be used. This pressure is reduced in the private Fock version of the code, and it is essentially eliminated in the shared Fock version. Overall, for the 2 nm dataset, the shared Fock code runs about six

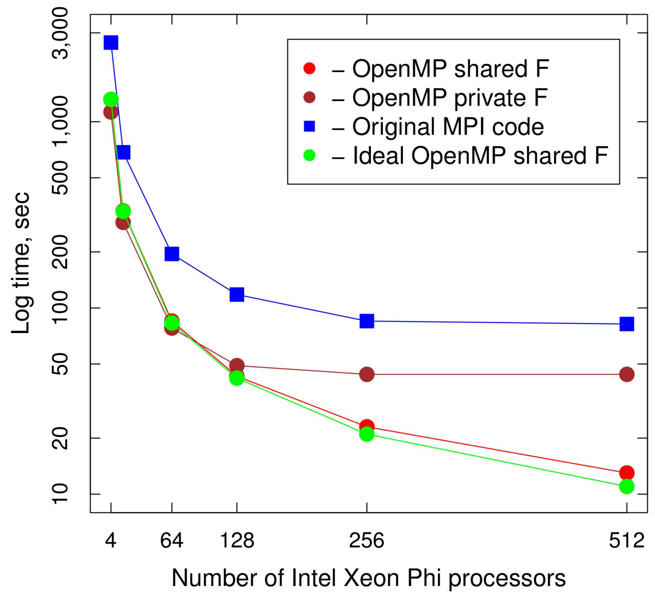

Figure 6: Multi-node scalability of the Private Fock and the Shared Fock hybrid MPI-OpenMP and the MPI-only stock GAMESS codes on the Theta machine with the 2.0 nm dataset. The quad-cache cluster-memory mode was used for all data points.

Table 3: Parallel efficiency of the three different HF algorithms using 2.0 nm dataset

# Nodes	Time-to-solution, s			Parallel efficiency, %		
	MPI[a]	Pr.F.[b]	Sh.F.[c]	MPI[a]	Pr.F.[b]	Sh.F.[c]
4	2661	1128	1318	100	100	100
16	685	288	332	97	98	99
64	195	78	85	85	90	97
128	118	49	43	70	72	96
256	85	44	23	49	40	90
512	82	44	13	25	20	79

[a] MPI-only SCF code [b] Private Fock SCF code
[c] Shared Fock SCF code

times faster than stock GAMESS on 512 Xeon Phi processors. It resulted from the better load balance of the shared Fock algorithm that uses all four shell indices – two are used in MPI and two are used in OpenMP workload distribution. The actual timings and efficiencies are listed in Table 3.

Figure 7 shows the behavior of the shared Fock version of GAMESS for the 5 nm dataset. It is the largest dataset we could fit in memory on Theta. Since we run on 4 MPI ranks the memory footprint is approximately 208 GB per node. This figure shows good scaling of the code up to 3,000 Xeon Phi nodes, which is equal to 192,000 cores (64 cores per node).

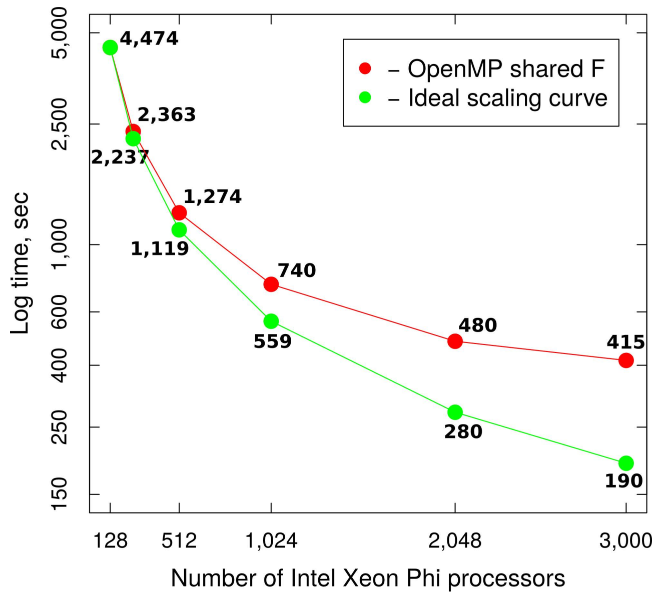

Figure 7: Scalability of the Shared Fock hybrid MPI-OpenMP version of GAMESS on the Theta machine for the 5.0 nm (i.e. large) dataset in quadrant cache mode on 3,000 Intel® Xeon Phi^{TM} processors. The results here are for 4 MPI ranks per node with 64 threads per rank, giving full saturation (in terms of hardware threads) on every Intel® Xeon Phi^{TM} node. For each point in the figure, we show the time in seconds.

7 CONCLUSION

In this paper, conversion of the MPI-only GAMESS HF code to hybrid MPI-OpenMP versions is described. The resulting hybrid implementations are benchmarked to exhibit improvements in the time-to-solution and memory footprint compared to the original MPI-only version. The code design decisions taken here were justified and implemented in a systematic way. Focus was placed on sharing the two primary (memory consuming) objects, the density and Fock matrices, in the SCF loop among the computation units. To the best of our knowledge, having a shared Fock matrix is an unique feature of our implementation. Indeed, this is absent in all other threaded HF codes known to us.

We have discussed two new HF implementations, each of which maintains full functionality of the underlying GAMESS code. In the first version, the density matrix was shared across threads, while the Fock matrix was kept private. The second version leveraged the first step, and focused entirely on making the Fock matrix a shared object. As a result, the memory footprint of the original code was lowered systematically while improving cache utilization and time-to-solution. Clearly, we have taken only the first steps towards an efficient hybrid HF implementation in GAMESS. In future work, we plan to tune our hybrid OpenMP/MPI code more thoroughly.

Our new hybrid MPI/OpenMP codes significantly outperform the official stock MPI-only code in GAMESS. Our best case implementation has about 200 times smaller memory footprint and runs up to 6 times faster than the original MPI-only version. Both our hybrid versions also have better scalability with respect to cores and nodes on single node and multi-node Intel Xeon Phi systems respectively.

It is also noted that the code optimizations reported in this paper are expected to be applicable to all previous and future generations of Intel Xeon Phi processors, as well as beneficial on the Intel Xeon multicore platform. The fact that the code already scales well on a large number of second generation Intel Xeon Phi processors enables us to help bring the promise of the "many-core" philosophy to the large scientific community that has long benefited from the extensive functionality of the GAMESS code. Like the MPI-only version, the hybrid versions of GAMESS can be deployed on systems ranging from a single desktop to large supercomputers. In addition, the hybrid codes offer enhanced configurability and parallel granularity.

Finally, the lessons learned here are applicable to virtually any code that handles non-linear partial differential equations using a matrix representation. In this paper, we treat the problem of assembling a matrix in parallel subject to highly non-regular data dependencies. Indeed, a variety of methods, such as Unrestricted Hartree Fock (UHF), Generalized Valence Bond (GVB), Density Functional Theory (DFT), and Coupled Perturbed Hartree-Fock (CPHF), all have this structure. The implementation of these methods can therefore directly benefit from this work. Beyond quantum chemistry, we note, the SCF approach shares much in common with generic non-linear solvers. We therefore conclude that the strategies discussed in this work are directly applicable to computer programs encountered in other areas of science.

ACKNOWLEDGMENTS

This research used the resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy (DOE) Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. Dr. Kristopher Keipert and Prof. Mark S. Gordon acknowledge the support of a Department of Energy Exascale Computing Project grant to the Ames Laboratory. We thank the Intel® Parallel Computing Centers program for funding. The authors would like to thank the RSC Technologies staff for the discussions and help.

REFERENCES

[1] 2017. Argonne National Laboratory Leadership Computing Facility. (2017). http://www.alcf.anl.gov/

[2] 2017. The General Atomic and Molecular Electronic Structure System (GAMESS) is a general ab initio quantum chemistry package. (2017). http://www.msg.ameslab.gov/gamess/index.html

[3] 2017. U.S. Department of Energy Office of Science Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. (2017). http://www.doeleadershipcomputing.org/

[4] Yuri Alexeev, Ricky A Kendall, and Mark S Gordon. 2002. The distributed data SCF. *Computer Physics Communications* 143, 1 (2002), 69–82.

[5] Yuri Alexeev, Ashutosh Mahajan, Sven Leyffer, Graham Fletcher, and Dmitri G Fedorov. 2012. Heuristic static load-balancing algorithm applied to the fragment molecular orbital method. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE, 1–13.

[6] Yuri Alexeev, Michael W Schmidt, Theresa L Windus, and Mark S Gordon. 2007. A parallel distributed data CPHF algorithm for analytic Hessians. *Journal of Computational Chemistry* 28, 10 (2007), 1685–1694.

[7] Edoardo Apra, Michael Klemm, and Karol Kowalski. 2014. Efficient implementation of many-body quantum chemical methods on the Intel® Xeon Phi co-processor. In *2014 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE Press, 674–684.

[8] Andrey Asadchev and Mark S Gordon. 2012. New multithreaded hybrid CPU/GPU approach to Hartree–Fock. *Journal of Chemical Theory and Computation* 8, 11 (2012), 4166–4176.

[9] Edmond Chow, Xing Liu, Sanchit Misra, Marat Dukhan, Mikhail Smelyanskiy, Jeff R Hammond, Yunfei Du, Xiang-Ke Liao, and Pradeep Dubey. 2015. Scaling up Hartree–Fock calculations on Tianhe-2. *International Journal of High Performance Computing Applications* (2015), 85–102.

[10] Edmond Chow, Xing Liu, Mikhail Smelyanskiy, and Jeff R Hammond. 2015. Parallel scalability of Hartree–Fock calculations. *The Journal of Chemical Physics* 142, 10 (2015), 104103.

[11] Dmitri G Fedorov and Kazuo Kitaura. 2007. Extending the power of quantum chemistry to large systems with the fragment molecular orbital method. *The Journal of Physical Chemistry A* 111, 30 (2007), 6904–6914.

[12] D G Fedorov and K Kitaura (Eds.). 2009. *The Fragment Molecular Orbital Method: Practical Applications to Large Molecular Systems.* CRC Press, Boca Raton, FL, USA.

[13] Graham D Fletcher, Michael W Schmidt, Brett M Bode, and Mark S Gordon. 2000. The distributed data interface in GAMESS. *Computer Physics Communications* 128, 1 (2000), 190–200.

[14] Ian T Foster, Jeffrey L Tilson, Albert F Wagner, Ron L Shepard, Robert J Harrison, Rick A Kendall, and Rik J Littlefield. 1996. Toward high-performance computational chemistry: I. Scalable Fock matrix construction algorithms. *Journal of Computational Chemistry* 17, 1 (1996), 109–123.

[15] Mark S Gordon and Michael W. Schmidt. 2005. Advances in electronic structure theory: GAMESS a decade later. In *Theory and Applications of Computational Chemistry: the first forty years*, CE Dykstra, G. Frenking, KS Kim, and GE Scuseria (Eds.). Elsevier, Amsterdam, 1167–1189.

[16] Robert J Harrison, Martyn F Guest, Rick A Kendall, David E Bernholdt, Adrian T Wong, Mark Stave, James L Anchell, Anthony C Hess, Rik J Littlefield, George L Fann, and others. 1996. Toward high-performance computational chemistry: II A Scalable Self Consitent Field Program. *Journal of Computational Chemistry* 17, 1 (1996), 124–132.

[17] Kazuya Ishimura, Kei Kuramoto, Yasuhiro Ikuta, and Shi-aki Hyodo. 2010. MPI/OpenMP hybrid parallel algorithm for hartree- fock calculations. *Journal of Chemical Theory and Computation* 6, 4 (2010), 1075–1080.

[18] Curtis L. Janssen and Ida M.B. Nielsen. 2008. *Parallel computing in quantum chemistry.* CRC Press, Boca Raton, FL, USA.

[19] Shigeki Kawai, Andrea Benassi, Enrico Gnecco, Hajo Söde, Rémy Pawlak, Xinliang Feng, Klaus Müllen, Daniele Passerone, Carlo A Pignedoli, Pascal Ruffieux, and others. 2016. Superlubricity of graphene nanoribbons on gold surfaces. *Science* 351, 6276 (2016), 957–961.

[20] Xing Liu, Anup Patel, and Edmond Chow. 2014. A new scalable parallel algorithm for Fock matrix construction. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International.* IEEE, 902–914.

[21] Vladimir Mironov, Maria Khrenova, and Alexander Moskovsky. 2015. On Quantum Chemistry Code Adaptation for RSC PetaStream Architecture. In *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings.* Springer, 113–121.

[22] David Ozog, Amir Kamil, Yili Zheng, Paul Hargrove, Jeff R Hammond, Allen Malony, Wibe de Jong, and Kathy Yelick. 2016. A Hartree-Fock application using UPC++ and the new DArray library. In *Parallel and Distributed Processing Symposium, 2016 IEEE International.* IEEE, 453–462.

[23] Spencer Pruitt. 2016. private communication. (2016).

[24] Michael W Schmidt, Kim K Baldridge, Jerry A Boatz, Steven T Elbert, Mark S Gordon, Jan H Jensen, Shiro Koseki, Nikita Matsunaga, Kiet A Nguyen, Shujun Su, and others. 1993. General Atomic and Molecular Electronic Structure System. *Journal of Computational Chemistry* 14, 11 (1993), 1347–1363.

[25] Ivan S Ufimtsev and Todd J Martinez. 2008. Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. *Journal of Chemical Theory and Computation* 4, 2 (2008), 222–231.

[26] Ivan S Ufimtsev and Todd J Martinez. 2009. Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation. *Journal of Chemical Theory and Computation* 5, 4 (2009), 1004–1015.

[27] Hiroaki Umeda, Yuichi Inadomi, Toshio Watanabe, Toru Yagi, Takayoshi Ishimoto, Tsutomu Ikegami, Hiroto Tadano, Tetsuya Sakurai, and Umpei Nagashima. 2010. Parallel Fock matrix construction with distributed shared memory model for the FMO-MO method. *Journal of Computational Chemistry* 31, 13 (2010), 2381–2388.

[28] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, and others. 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477–1489.

[29] Karl A Wilkinson, Paul Sherwood, Martyn F Guest, and Kevin J Naidoo. 2011. Acceleration of the GAMESS-UK electronic structure package on graphical processing units. *Journal of Computational Chemistry* 32, 10 (2011), 2313–2318.

A ARTIFACT DESCRIPTION: AN EFFICIENT MPI/OPENMP PARALLELIZATION OF THE HARTREE-FOCK METHOD FOR THE SECOND GENERATION OF INTEL® XEON PHI^{TM} PROCESSOR

A.1 Abstract

This description contains all the information needed to run the simulations described in the SC17 paper "An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel® Xeon Phi^{TM} processor". More precisely, we explain how to compile and run GAMESS simulations to reproduce results presented in Section 6 of the paper. The PDB files, GAMESS inputs, and submission scripts used in this work can be downloaded from the Git repository available online[2].

A.2 Description

A.2.1 Check-list (artifact meta information). Fill in whatever is applicable with some informal keywords and remove the rest

- **Algorithm:** Self Consistent Field (SCF) algorithm in Hartree-Fock (HF) method
- **Program:** GAMESS binary, Fortran and C libraries
- **Compilation:** Intel® Parallel Studio XE 2016 v3.210
- **Data set:** graphene bilayer systems from 0.5 nm to 5 nm; details are in the main paper section 5 and in Table 4
- **Run-time environment:** when running on Cray XC40 the following modules were loaded:
  - craype/2.5.9
  - PrgEnv-intel/6.0.3
  - craype-mic-knl
- **Hardware:** all single node benchmarked were run on JLSE cluster on Intel® Xeon Phi^{TM} 7210 processor; all multi-node benchmarks were run on Cray XC40 with up to 3,000 Intel® Xeon Phi^{TM} 7230 processors
- **Experiment customization:** varied number of MPI ranks from 1 to 12,000 and number of OpenMP threads from 1 to 256 on single node.
- **Publicly available?:** yes (partially)

A.2.2 How software can be obtained (if available). In this work, we used GAMESS version dated August 18, 2016 Revision 1. The original GAMESS code can be downloaded (at no cost) from the official GAMESS website (registration required)[3].

Patches for GAMESS developed in this work can be obtained from the authors by request.

A.2.3 Hardware dependencies. All measurements in this work were performed on the 2nd generation of Intel® Xeon Phi^{TM} processor. However, the same benchmarks can be run on any other architectures.

A.2.4 Software dependencies. C (C99 standard) compiler, Fortran 95 compatible compiler with OpenMP 3.0 support, MPI library. The code has been extensively tested only for Intel® Parallel Studio XE 2016 update 3 compilers and libraries. MKL BLAS library was used to link GAMESS, but it does not affect the performance of the SCF code and thus, it is optional.

[2]https://github.com/gms-bbg/gamess-papers, folder Mironov_Vladimir-2017-SC17
[3]http://www.msg.ameslab.gov/gamess/download.html

A.2.5 Datasets. All structures of chemical systems and corresponding input files used for benchmarking code can be downloaded from the Git repository[2]. They are easily reconfigurable bi-layer graphene systems. The problem size (computation time, memory footprint) depends on the number of basis functions for the system. These numbers are provided in Table 4. The basis set used in all calculations is 6-31G(d).

Table 4: Chemical systems used in benchmarks and their size characteristics. BF stands for basis function.

Name	# atoms	# shells	# basis functions
0.5 nm	44	176	660
1 nm	120	480	1,800
1.5 nm	220	880	3,300
2 nm	356	1,424	5,340
5 nm	2,016	8,064	30,240

A.3 Installation

We followed the standard installation procedure outlined in the file `PROG.DOC` in GAMESS root directory. GAMESS uses a custom build configuration system. The first step is configuration of the `install.info` file. To perform the basic configuration one need to run `${GMS_HOME}/config` script and specify compilers and libraries for the compilation. When the script asks whether to compile GAMESS with LIBCCHEM one need to refuse. After that the `install.info` file can be edited directly to get the desired configuration. After setup is finished, GAMESS compilation can be done with `make` command. At successful conclusion, the file `gamess.$(VERNO).x` will appear in GAMESS home directory, where `$(VERNO)` is a variable specified at basic configuration step with `${GMS_HOME}/config` script. `VERNO="00"` by default.

We used two different Intel® Xeon Phi^{TM}systems: JLSE and XC40. The actual `install.info` configurations are accessible at Git repository in folders `JLSE` and `CRAYXC40`. The key parameters of the `install.info` file for both clusters are summarized in Table 5.

Moreover, we manually added the flag `-xMIC-AVX512` to the compilation line in `${GMS_HOME}/comp` script and added `-mkl` flag to the link line in `${GMS_HOME}/lked` script. On Cray XC40 system we also modified `${GMS_HOME}/comp`, `${GMS_HOME}/compall`, `${GMS_HOME}/lked`, and `${GMS_HOME}/ddi/compddi` scripts to add a new target "cray-xc". The modified files are accessible at Git repository in folders `JLSE` and `CRAYXC40`.

For DDI library, we used an experimental version of software to run all benchmarks for the single node Intel® Xeon Phi^{TM} performance on JLSE cluster. This DDI library uses one-sided communication features of MPI-3 which does not spawn data servers. On the Cray XC40 system, we used the standard DDI library.

A.4 Experiment workflow

We used the standard workflow of the experiment: compile code and run it for different Intel® Xeon Phi^{TM} system. We varied number of MPI ranks from 1 to 3,000, number of threads from 1 to 256 per rank, and varied different Xeon Phi^{TM} clustering and memory modes.

A.5 Evaluation and expected result

We ran GAMESS on the Joint Laboratory for System Evaluation (JLSE) cluster using this submission line for OpenMP code:

```
mpirun -n $2 \
  -env OMP_NUM_THREADS $3 \
  -env I_MPI_SHM_LMT shm \
  -env KMP_STACKSIZE 200m \
  -env I_MPI_PIN_DOMAIN auto \
  -env KMP_AFFINITY verbose \
  -env <GAMESS options> gamess.00.x
```

On the XC40 system, we used the following submission line using Cobalt queuing system for OpenMP code:

```
rpn=4
allranks=$((COBALT_JOBSIZE*rpn))
aprun -n $allranks -N $rpn \
  -d $threads -cc depth -j 4 \
  --env OMP_NUM_THREADS=64 \
  --env I_MPI_PIN_DOMAIN=64:compact \
  --env KMP_STACKSIZE=2000m \
  --env <GAMESS options> gamess.00.x
```

For MPI code, we used this submission line:

```
rpn=4
allranks=$((COBALT_JOBSIZE*rpn))
aprun -n $allranks -N $rpn \
  -cc depth -j 4 \
  --env I_MPI_PIN_DOMAIN=64:compact \
  --env <GAMESS options> gamess.00.x
```

The submission and run scripts are accessible at Git repository in folders JLSE and CRAYXC40.

The results of the computation are printed to STDOUT or redirected to the file specified in submission/run script. Time for the SCF part of code can be obtained with the following command:

```
$ grep "TIME TO FORM FOCK" <logfile> \
    | awk '{print $9}'
```

It will return the time in seconds for the Fock matrix construction step.

Table 5: Key configuration parameters of the install.info file for the supercomputers used in this work

Parameter	JLSE	Cray XC40
GMS_TARGET	linux64	cray-xc
GMS_FORTRAN	ifort	ftn
GMS_MATHLIB	mkl	mkl
GMS_DDI_COMM	mpi	mpi
GMS_MPI_LIB	impi	impi
GMS_LIBCCHEM	false	false
GMS_PHI	true	true
GMS_SHMTYPE	sysv	posix
GMS_OPENMP	true	true

A.6 Experiment customization

During the experiment we changed the following runtime parameters:

- The total number of MPI processes with batch script parameters
- The number of OpenMP threads per MPI process by adjusting OMP_NUM_THREADS environmental variable
- Memory and clustering modes of Intel® Xeon Phi^{TM} nodes (BIOS parameters, restart is required)
- The environmental variables for Intel MPI and Intel OpenMP libraries (KMP_AFFINITY, I_MPI_PIN_DOMAIN)

A.7 Notes

Some of the GAMESS timer routines use CPU time instead of wall clock time without informing the user about it. Timing results from these routines will be incorrect for multithreaded code. Therefore, we used omp_get_wtime() function to measure the performance of OpenMP code.