Parallel Implementation and Performance Optimization of the Configuration-Interaction Method

Hongzhang Shan
Computational Research Division
Lawrence Berkeley Laboratory
Berkeley, CA, 94720
hshan@lbl.gov

Kenneth McElvain
Department of Physics
University of California, Berkeley
Berkeley, CA 94720
kenmcelvain@me.com

Calvin W. Johnson
Department of Physics
San Diego State University
San Diego, CA 92182
cjohnson@mail.sdsu.edu

Samuel Williams
Computational Research Division
Lawrence Berkeley Laboratory
Berkeley, CA, 94720
swwilliams@lbl.gov

W. Erich Ormand
Department of Physics
Lawrence Livermore Laboratory
Livermore, CA 94551
ormand1@llnl.gov

ABSTRACT

The *configuration-interaction* (CI) method, long a popular approach to describe quantum many-body systems, is cast as a very large sparse matrix eigenpair problem with matrices whose dimension can exceed one billion. Such formulations place high demands on memory capacity and memory bandwidth — two quantities at a premium today. In this paper, we describe an efficient, scalable implementation, BIGSTICK, which, by factorizing both the basis and the interaction into two levels, can reconstruct the nonzero matrix elements on the fly, reduce the memory requirements by one or two orders of magnitude, and enable researchers to trade reduced resources for increased computational time. We optimize BIGSTICK on two leading HPC platforms — the Cray XC30 and the IBM Blue Gene/Q. Specifically, we not only develop an empirically-driven load balancing strategy that can evenly distribute the matrix-vector multiplication across 256K threads, we also developed techniques that improve the performance of the Lanczos reorthogonalization. Combined, these optimizations improved performance by 1.3-8× depending on platform and configuration.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*; D.2 [**Software Engineering**]: Metrics—*Performance measures*

Keywords

BIGSTICK, configuration-interaction, Lanczos, eigenvalue, reorthogonalization, load balancing, performance, scalability

SC '15, November 15-20, 2015, Austin, TX, USA
© 2017 ACM. ISBN 978-1-4503-3723-6/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807618

1. BACKGROUND

Atoms and their nuclei are quantum *many-body* systems. We describe such systems by the non-relativistic many-body Schrödinger equation $\mathbf{H}\Psi(r_1, r_2, \ldots, r_A) = E\Psi$, with

$$\mathbf{H} = \sum_{i=1}^{A} -\frac{\hbar^2}{2M}\nabla_i^2 + \sum_{i<j} V(|r_i - r_j|). \tag{1}$$

While the Schrödinger equation is linear, it is a partial differential equation with $6A$ independent variables (including spin) for A particles, and is all but impossible to solve for more than four particles.

The many-body wavefunction is important for many arenas. For example there is strong evidence that non-baryonic dark matter is a significant component of the cosmos [20, 30]. Detection and characterization of exotic particles such as dark matter often hinge upon their interactions with nuclei. As such, the spin-dependent elastic scattering rate is proportional to the nuclear spin response [15]

$$\frac{a_T}{2J+1}|\langle\Psi||\vec{S}_p \pm \vec{S}_n||\Psi\rangle|^2, \tag{2}$$

where Ψ is the nuclear ground state wave function, and the double-bars denote a reduced matrix element [13]. The $+$ sign is for the isoscalar response and $-$ is for isovector, while a_T is the coupling constant between dark matter particles and nucleons (and is likely different for isoscalar, $T = 0$ and isoscalar, $T = 1$ responses). By measuring the response for different targets we can in principle determine the value of a_T. Among the possible targets for dark matter detectors are naturally occurring Xenon [2] and Cesium [17] and could be used in accelerator or terrestrial experiments [14, 16, 30]. We are applying our optimized BIGSTICK to a large portfolio of proposed dark matter target isotopes and couplings in order to identify the most promising candidates. Some early results will be presented in Section 10.

To calculate the nuclear response, we find approximate solutions to Eq. 1 by rewriting it as a matrix equation. Using a finite set of basis functions $\{\Phi_\alpha(r_1, r_2, \ldots, r_A)\}$ we expand the wave function $\Psi = \sum_\alpha c_\alpha \Phi_\alpha$, and solve for the coefficient c_α and the energy E by computing the matrix elements $H_{\alpha\beta} = \Phi_\alpha^\dagger \mathbf{H} \Phi_\beta$ and finding the eigenpairs.

We can either choose basis states which are already highly correlated, but of which we need only a few, or we can build up correlations from many simple basis states. Although both paths have their advantages, we follow the latter, using so-called Slater determinants [27, 23], antisymmeterized products of single-particle states, that is $\Phi_\alpha(r_1, r_2, \ldots, r_A) = \mathcal{A}\ \phi_1(r_1)\ \phi_2(r_2) \ldots \phi_A(r_A)$, where $\mathcal{A}$ indicates a sum over all antisymmetric permutations of the coordinates $r_1, r_2, \ldots$. If the *single-particle* state $\{\phi_i(r)\}$ are orthonormal, the resulting Slater determinants are automatically also orthonormal, and one can handle them using fermion creation and annihilation operators. This approach is often called the *configuration-interaction* (CI) method [11, 27, 6, 5, 9]. As we are most often interested in low-lying eigenpairs, we use the Lanczos algorithm [33, 9].

This path is not without a price. Many systems require millions or even billions of basis states for appropriate description; below we will describe systems with dimensions up to 9 billion. The resultant matrices are very sparse with a density of roughly 10^{-6}. Nevertheless, this still leads to a very large number of nonzero matrix elements: if the dimension were 10^9, there could be more than 10^{12} nonzero matrix elements constituting at least O(10) TB of data. Thus, we have three challenging computational issues: (1) efficiently finding the location and values of those one-in-a-million nonzero matrix elements $H_{\alpha\beta}$; (2) efficient storage or recall of those nonzero matrix elements; and (3) efficient load-balancing when solving the eigenvalue problem.

Although (1) and (3) could be addressed via direct storage of a sparse matrix, either on disk [33, 3], where the performance bottleneck is I/O time, or in DRAM [28], where the performance bottleneck is the sheer amount of DRAM needed, we exploit native properties of the physical systems, namely conservation laws we describe in the next section, to allow us to 'factorize' the problem and retrieve the nonzero matrix element on-the-fly. Such an approach greatly reduces the time spent identifying the nonzero matrix elements and reduces the memory needed to store the nonzero matrix elements by a factor of 10 or more. Furthermore factorization allows one to efficiently compute the number of nonzero matrix elements in a block of work, which in turn allows one to effectively distribute the work. While several other codes utilize similar factorization algorithms [10, 8, 4, 25, 32], we work with the BIGSTICK configuration-interaction code [19], focusing on efficient load-balancing of matrix-vector multiplication and of reorthogonalization critical for BIGSTICK to be successful on the large-scale HPC platforms.

2. FACTORIZATION IN CONFIGURATION-INTERACTION CALCULATIONS

The central problem of configuration-interaction calculations is finding extremal eigenpairs of a large, sparse matrix $\mathbf{H}$ (the Hamiltonian matrix). Due to physical conservation laws, the matrix $\mathbf{H}$ has a block structure we can exploit. The conservation law we exploit is expressed by Noether's theorem in $[\mathbf{H}, \mathbf{J}_z] = 0$, where the matrix $\mathbf{J}_z$ represents the component of angular momentum in the z-direction. The eigenvalues of $\mathbf{J}_z$ are highly degenerate, with each eigenvalue labeling a subspace (an irreducible representation or an irrep in group theory), and because of conservation the Hamiltonian matrix $\mathbf{H}$ is block-diagonal in the irreps of $\mathbf{J}_z$. It is easy to construct this block-diagonal structure because the group defined by J_z is Abelian, so that if a system is composed of several subsystem, each of which is an eigenstate of J_z, then the composite system is also an eigenstate of J_z, with its eigenvalue just the sum of the eigenvalues of the subsystems. In nuclei, we can choose our representations of each nucleon to be eigenstates of J_z with eigenvalue m_i (also called the *magnetic* quantum number), and then we choose our basis to be those states of A particles whose sum $M = m_1 + m_2 + m_3 + \ldots m_A$ is fixed; this is called an M-scheme basis. One can also use the matrix $\mathbf{J}^2$, which is the total angular momentum, to construct the so-called J-scheme basis. An M-scheme basis is larger in dimension than a J-scheme basis, sometimes a factor of ten larger, but is much easier to construct.

We can choose a block structure based upon subsystems and their J_z eigenvalues at any convenient level. We choose to describe each basis state as a simple product of a proton substate (with eigenvalue M_p) and a neutron substate (M_n) such that $M = M_p + M_n$. This leads to the concept of *factorization* [19]. First, given that total M is fixed, if we know M_p we know the associated $M_n = M - M_p$. Furthermore, we can generally assume that *all* proton substates with a given M_p not only can, but *must* combine with all neutron substates with $M_n = M - M_p$. This means we do not need to explicitly represent all the basis states, but only have to represent the constituent proton and neutron substates, and keep track of their respective eigenvalues M_p, M_n.

This factorization allows us to reduce the storage of the elements of the Hamiltonian matrix. Physically, the Hamiltonian operator can affect at most two nucleons at a time (generalizations to three nucleons are possible and have been implemented, but we leave those out to simplify the discussion). Thus we can classify the elements of the Hamiltonian as PP (two protons), NN (two neutrons), or PN (one proton and one neutron); a fourth kind of element are the single-particle energies or SPE which only contribute to the diagonal elements. The elements of the Hamiltonian matrix are still constrained by the block form dictated by conservation. So, as PP acts only on protons, neutrons are spectators, M_n cannot change, and hence M_p cannot change. Thus, while the entire Hamiltonian matrix is contained within a space defined by fixed M, the PP part of the Hamiltonian matrix can only be nonzero within blocks of fixed M_p. The same is true of course for NN, and there are similar constraints for PN and SPE.

The information controlling the reconstruction and application of the Hamiltonian matrix elements is organized into uniform data structures called *opbundles*, used to distribute the matvec work among the processes. The opbundle data structure contains information about the type of operation (PP, NN, etc.), the structure of loops over proton and neutron substates, and number of operations. BIGSTICK contains routines to subdivide opbundles among processes, allowing for improved load balancing at the expense of a slight increase in computation (an impediment to strong scaling).

3. PARALLEL IMPLEMENTATION

BIGSTICK, following other CI codes, uses the iterative Lanczos algorithm to compute the low-lying eigenstates. The algorithm is divided into two steps, the sparse matrix-vector multiplication (matvec) step followed by the reorthogonalization (reorth) step. Recall, the sparse matrix is never ex-

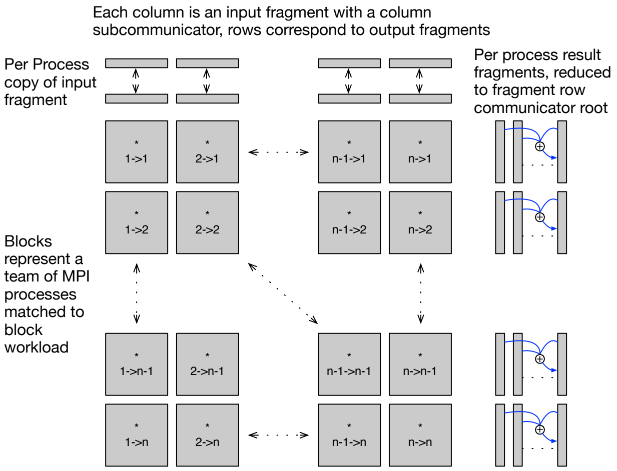
Each column is an input fragment with a column subcommunicator, rows correspond to output fragments
Per Process copy of input fragment
Per process result fragments, reduced to fragment row communicator root
1->1
2->1
n-1->1
n->1
1->2
2->2
n-1->2
n->2
Blocks represent a team of MPI processes matched to block workload
1->n-1
2->n-1
n-1->n-1
n->n-1
1->n
2->n
n-1->n
n->n

Figure 1: A schematic diagram of the matrix vector product flow. Blue lines indicate MPI communications.

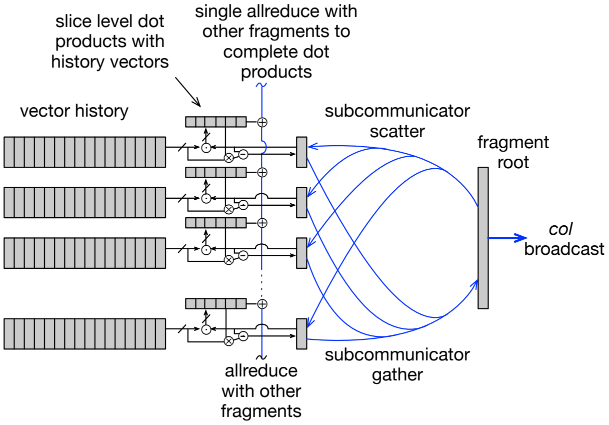
slice level dot products with history vectors
single allreduce with other fragments to complete dot products
vector history
subcommunicator scatter
fragment root
col broadcast
allreduce with other fragments
subcommunicator gather

Figure 2: A schematic diagram of the reorthogonalization flow. Blue lines indicate MPI communications.

plicitly formed or stored, but is continually reconstructed on the fly. When coupled with the number of nonzeros, the matrix-vector multiplication is the most important computation of the lanczos algorithm. To efficiently parallelize this operation, one is motivated to evenly distribute the nonzero Hamilton matrix elements and the Lanczos vectors across the MPI processes. To fulfill this purpose, the basis is divided into fragments based upon the proton substate eigenvalue M_p and upon a threshold predefined to govern the efficiency of factorization. Once the basis state vectors are divided into n *fragments*, the Hamiltonian matrix will be divided into $n \times n$ *blocks* correspondingly. Each block (i, j) includes all the jump operations from fragment j of the input basis state vector to fragment i of the output vector and the control information is contained in the opbundle data structures.

We can accurately predict the *number* of operations in a block (by operation we mean reconstruction and application of a matrix element) from the opbundle information. The MPI processes will be assigned to these blocks proportionally to the amount of work associated. All processes assigned to the same block form a team. The operations and associated data will be evenly distributed among the team members. Figure 1 shows the MPI processes divided into a two dimensional array of teams. Each MPI process now stores only one fragment of the input and output vectors, which is much less than the prior version where every MPI process kept complete vectors.

New MPI sub-communicators are formed in the block row and block column directions. The column, or *col* communicator is used to update the input fragment before beginning the matrix vector product. After each team computes its contribution to the output fragment, the *row* communicator is used to perform a reduction onto the rank 0 process of the communicator. An important detail is that the same MPI processes are assigned rank 0, which we will call the root, for all sub-communicators they are part of. There are total n root processes, one from each diagonal block.

The above algorithm using only the number of operations to distribute the work among the processes still causes significant load imbalance. Later in Section 7, we will introduce an empirically-driven load balancing algorithm, which differentiates the types of matrix elements and assigns different weight to their operation.

Once the matrix vector product step has finished the reduction, the next step is to reorthogonalize and normalize the result. The Lanczos algorithm produces a sequence of normalized vectors that are mathematically orthogonal to earlier vectors in the sequence. However, in practical implementations, the Lanczos vectors often lose their mutual orthogonality due to numerical errors. Therefore, a reorthogonalization process is needed to maintain the vector orthogonality.

To begin the process, the result fragments must be distributed to a number of MPI processes which will store slices of the fragments and perform the computations. To simultaneously balance the workload and storage requirements of reorthogonalization, the distribution of slices is done via a third sub-communicator which we call the *reorth* communicator. This sub-communicator again shares the same root as the row and column ones. The member processes are assigned to the corresponding fragment in a manner akin to round-robin but proportionally to the fragment size.

The flow, as seen in Figure 2, is illustrated for one fragment. The root process scatters the data to the slice MPI processes over the *reorth* communicator. A slice local dot-product is computed between the new slice and all the prior ones. These values are collected and reduced with a single global allreduce call to produce global dot-products against all prior vectors. Using the global dot-product results overlap is locally removed, the now orthogonalized new vector is normalized and each MPI process saves its slice of the vector. To begin the next Lanczos iteration, the slices for each fragment of the normalized vector are gathered back to the corresponding fragment root processes and then the column copies of the fragment are updated via a broadcast on the *col* communicator. BIGSTICK will iterate on these Lanczos iterations until convergence has been reached or a fixed number of iterations (e.g. 200) has been executed.

Memory capacity is at a premium in BIGSTICK. Duplicating tables among multiple MPI processes on a shared memory node architecture is inefficient. As such, BIGSTICK uses OpenMP to maintain parallelism while conserving memory by sharing the tables used in the construction of matrix elements. For simple loops, BIGSTICK applies the "omp parallel do" directives directly. For complex loops like the core *matvec* phase where load balancing is difficult, BIGSTICK will compute the workload in advance and divide the work evenly while avoiding write conflicts between threads. BIGSTICK is developed in Fortran90 and uses both MPI and OpenMP to implement the parallelization.

4. RELATED WORK

The most closely related work to ours is MFDn [28], which implements the configuration interaction method by storing the nonzero matrix elements in memory instead of reconstructing them on the fly. Its performance has well been studied on several large-scale HPC platforms. P. Sternberg et al. described the code changes for MFDn and its improved performance on NERSC's XT4 (Franklin) [28]. Since then, its performance had been studied on different multicore platforms [1]. Recently, the matrix construction part was ported to the NVIDIA GPU architecture and evaluated on the Oak Ridge's XK7 (Titan) [26]. As both BIGSTICK and MFDn use M-scheme bases, the information in their Hamiltonian matrices and the eigenpair solutions are identical. The ordering of the basis, and thus of matrix elements and of vector coefficients, are quite different: BIGSTICK groups the basis by M_p to exploit factorization, whereas MFDn orders the basis to make distribution of the non-zero elements of $\mathbf{H}$ as uniform as possible.

Other codes, such as OXBASH [3], ANTOINE [10], NATHAN [8], NuShellX [4], and EICODE [25], have not been evaluated or ported to large-scale, distributed-memory parallel supercomputers. The preliminary performance of BIGSTICK on NERSC's Cray XE6 (Hopper) had been reported in [19]. Since then, both the code design and the performance have been improved substantially. BIGSTICK can now solve problems with basis sizes exceeding 9 billion.

The design and performance of MPI collective operations, which are heavily used in BIGSTICK and performance critical, have been extensively studied. R. Thakur and W. Gropp described the algorithms used by MPICH [31]. Some specific implementations on the IBM BG/Q platform have been discussed in [22]. K. Kandalla et al. discussed how to develop the topology-aware algorithms for Infiniband clusters [21]. For the Aries network used on the Cray XC30 platform, N. Jain et al. [18] found that using random job placement can often avoid hot-spots and deliver better performance. Similarly, K. Kandalla [7] found that increasing the average communication distance between nodes may improve the performance when the system utilization is over 80%. These results is in contrast to the conventional wisdom that usually advocates to pack the placement of a job as close as possible. In our study, we have a similar discovery.

5. EXPERIMENTAL SETUP

5.1 Platforms

In this paper, we evaluate and optimize BIGSTICK performance on two HPC computing platforms.

Table 1: Test Problem Characteristics

	Frozen Core	Valence Protons	Valence Neutrons	Basis Dimension	Nonzero Elements
^{132}Cs	^{100}Sn	5	27	1 Billion	2.39×10^{12}
^{112}Xe	^{100}Sn	4	8	9.3 Billion	1.43×10^{13}

Edison is a Cray XC30 system located at NERSC [12]. It is comprised of 5,576 compute nodes, each of which contains two 12-core Intel Ivy Bridge out-of-order superscalar processors running at 2.4 GHz, and is connected by Cray's Aries (Dragonfly) network. On Edison, the third rank of the dragonfly is substantially tapered and, as all experiments which run in production mode, there is no control over job placement. Each core includes private 32KB L1 and 256KB L2 caches, and each processor includes a shared 30MB L3 cache. Nominal STREAM [29] bandwidth to DRAM is roughly 103 GB/s per compute node.

Mira is an IBM Blue Gene/Q located at Argonne National Laboratory [24]. Mira is composed of 49,152 compute nodes, each of which includes 16 multithreaded PowerPC A2 cores for user code and one additional core for operating system services. Each in-order, dual-issue core runs at 1.6 GHz and supports four threads. Unlike Ivy Bridge, at least 2 threads per core (32 threads per node) are required to efficiently utilize the A2 processor; we run with the full 64 threads supported by each node. The cache hierarchy is very different from the Ivy Bridge processor in that each core only has a small 16KB L1 cache, while all cores on a node share a 32MB L2 cache. Nodes are interconnected using IBM's high-performance proprietary network in a 5D torus. The STREAM bandwidth is approximately 26GB/s.

Superficially, a processor on Edison should provide superior performance and its out-of-order nature mitigates the complexity of opbundle operation processing. Conversely, Mira's 5D torus should provide superior performance on collective operations.

5.2 Test Problems

In this paper, we use two test problems — ^{132}Cs and ^{112}Xe — listed in Table 1. In both cases, we use a frozen core of ^{100}Sn. This provides ^{132}Cs with 5 valence protons and 27 neutrons and ^{112}Xe with 4 valence protons and 8 neutrons. The dimension of the resultant problems are 1 and 9.3 billion respectively. Effectively, the ^{112}Xe problem has 14 trillion nonzeros. BIGSTICK mitigates the storage and data movement bottlenecks associated with such large matrices allowing us to run problems for which a stored matrix representation might require 150TB using as few as 32 nodes. Nevertheless, when encoded in opbundles, there can be as few as about 5000 opbundles for these problems. As such, this finite, coarse-grained parallelism (opbundles) can impose an impediment to strong scaling to large concurrencies.

Our test problems were selected based on two criteria. First, they were of an appropriate size for our computational resources (memory, CPU-hours, etc...). Second, they are similar to isotopes being proposed for dark matter detector experiments. Thus, in this paper, we use these isotopes to improve the performance of BIGSTICK in preparation for the use of BIGSTICK in the context of nuclear science.

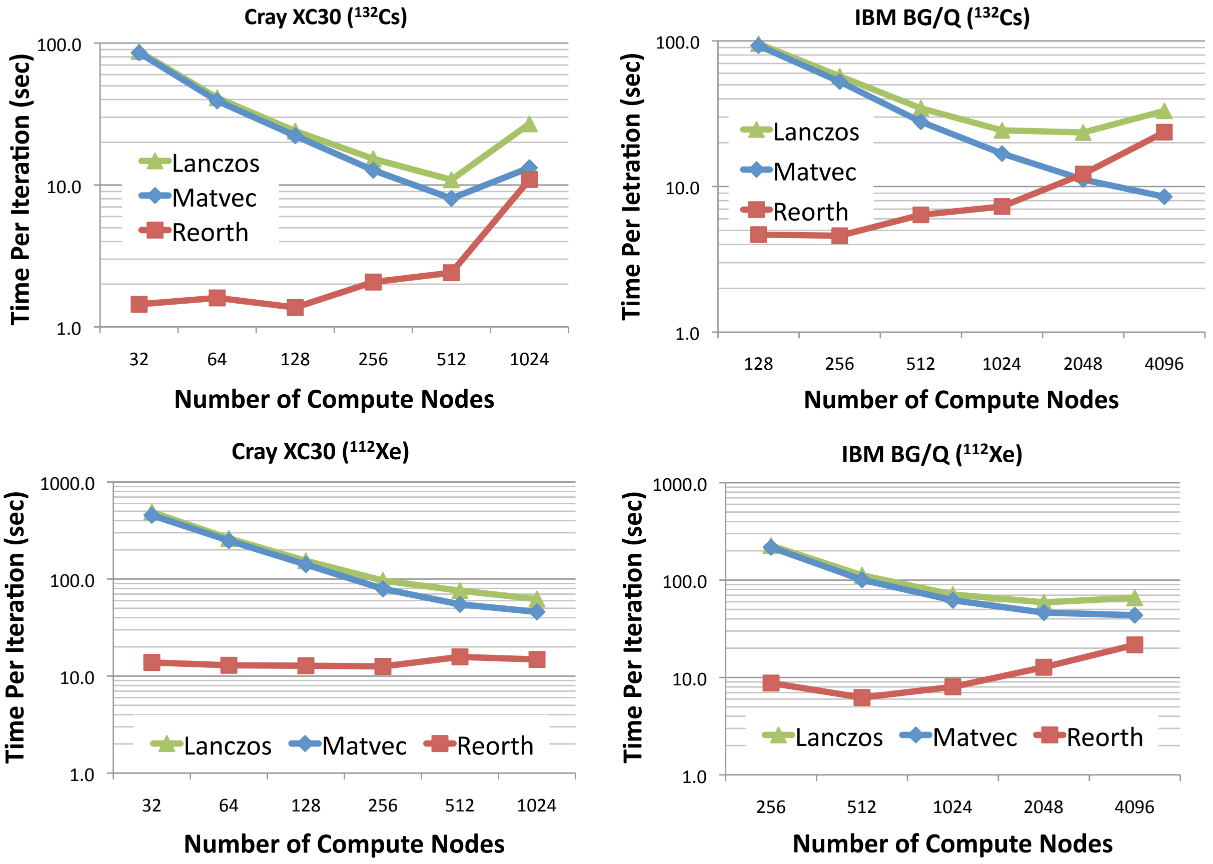

Figure 3: Baseline performance for ^{132}Cs and ^{112}Xe when strong scaling on the Cray XC30 and the IBM BG/Q. Lanczos iteration time is broken down into Matvec and Reorth time. On the Cray XC30, each node runs 4 MPI processes with 6 OpenMP threads for ^{132}Cs and 2 MPI processes with 12 OpenMP threads for ^{112}Xe. On the IBM BG/Q, each node runs 4 MPI processes with 16 OpenMP threads for ^{132}Cs and 1 MPI process with 64 OpenMP threads for ^{112}Xe. Note the log-log scale.

6. BASELINE PERFORMANCE

Before discussing performance optimizations, we present the baseline strong scaling performance of BIGSTICK using the ^{132}Cs and ^{112}Xe test problems. Recall, the ^{112}Xe is roughly an order of magnitude larger. Figure 3 shows Lanczos time per iteration in the baseline MPI+OpenMP implementation as a function of the number of nodes on the Cray XC30 and the IBM BG/Q platforms. Moreover, we break Lanczos time down into its two major components — matrix-vector multiplication (matvec) and reorthogonalization (reorth).

For the ^{132}Cs problem on the Cray XC30 platform, Lanczos is dominated by matrix-vector multiplications and performance scales very well up to about 512 nodes. At 1K nodes case, Lanczos times increases substantially. There are two potential reasons for this spike. First, at this scale, the load imbalance begins to impede matvec performance. Second, the reorthogonalization time, which is insignificant earlier, increases quickly reaching about 40% of the Lanczos time. Similar performance characteristics is shown for ^{132}Cs on the IBM BG/Q platform. However, on the IBM BG/Q, matvec time continues to scale, while reorthogonalization sees a steady increase in time. At 2K nodes on BG/Q, reorthogonalization reaches parity with matvec and dominates it beyond reaching over 40% of the lanczos computing time at 4K nodes.

The ^{112}Xe results are shown at the Figure 3(bottom). Compared with ^{132}Cs, its basis vector size is about 10 times larger. On both machines, Lanczos is dominated by the MatVec execution times. Interestingly, there is no spike in reorthogonalization time on the XC30 while there remains a steady increase in reorthogonalization time on the BG/Q. In effect, the ^{112}Xe is larger and is easier to load balance. This ensures matvec both dominates the execution time and continues to scale to larger concurrencies.

Generally speaking, node-for-node, the XC30 (Edison) delivers 3-5× better performance (and comparable energy) than the IBM BG/Q (Mira) except for the 1K nodes case. For problems with substantial indirections and heavy memory usage, this should come as no surprise as the XC30 has roughly 4× the DRAM bandwidth.

The remaining focus of this paper is improving the performance and scalability of these machines in particular for the ^{132}Cs problem.

Table 2: Average time per operation in nanoseconds by opbundle type, platform, and problem (also used as empirically-determined weights)

^{132}Cs

	SPE	PP	NN	PN('B')	PN('F')
Cray XC30	2.26	1.01	0.38	2.66	0.51
IBM BG/Q	17.20	4.40	2.63	14.00	5.34

^{112}Xe

	SPE	PP	NN	PN('B')	PN('F')
Cray XC30	2.01	0.43	0.24	1.25	0.26
IBM BG/Q	17.18	0.93	0.70	2.08	1.40

7. EMPIRICALLY-DRIVEN MATVEC LOAD BALANCING

Figure 3 demonstrates a jump in matvec execution time for the ^{132}Cs on the XC30 at 1K nodes. Further analysis reveals that this is not only due to communication, but rather to increased load imbalance.

In BIGSTICK, the matrix-vector multiplication operations are governed by a data structure called opbundles, briefly described in Section 2. Opbundles are restricted to specific fragments of the input and output vectors in matvec. They are also restricted to a particular kind of operation: PP, NN, etc. The total number of opbundles of each type and the number of operations for each opbundle can be computed in advance. BIGSTICK evenly distributes the opbundles over MPI processes based on the number of operations without considering type.

We observe that for different types of opbundles, the time to perform a operation can differ substantially. This can be understood as arising from different ordering of the loops over proton substates and neutron substates. Such functional heterogeneity can lead to significant execution time differences and hence load imbalance.

To highlight this imbalance, Table 2 shows the average time per operation for different types of opbundles in nanoseconds (total time by opbundle divided by the number of operations). The usage of OpenMP threads helps to bring the average time down under 1 nanoseconds. We can see the average time per operation varies substantially across the opbundle types (SPE is nearly 7× more expensive than NN). We may thus develop a load balancing scheme based on opbundle type. We further refine this scheme by leveraging platform-specific information.

Originally, BIGSTICK had tried to incorporate this performance variation into consideration by assigning different weights for operations from different opbundle types. By assigning a different weight for each opbundle type, the performance can be improved significantly. However, we discovered that for the PN type, which consumes most of the matrix-vector multiplication times, assigning one weight is not sufficient; load imbalance persists, arising from the fact the operational time jumping from i to f states is not the same as from f to i. The information in these opbundles is identical, but again due to loop ordering the timing is not. Internally these opbundles are labeled as forward ('F') versus backwards ('B'). The average time per operation between these two subclasses can differ by up to 5×.

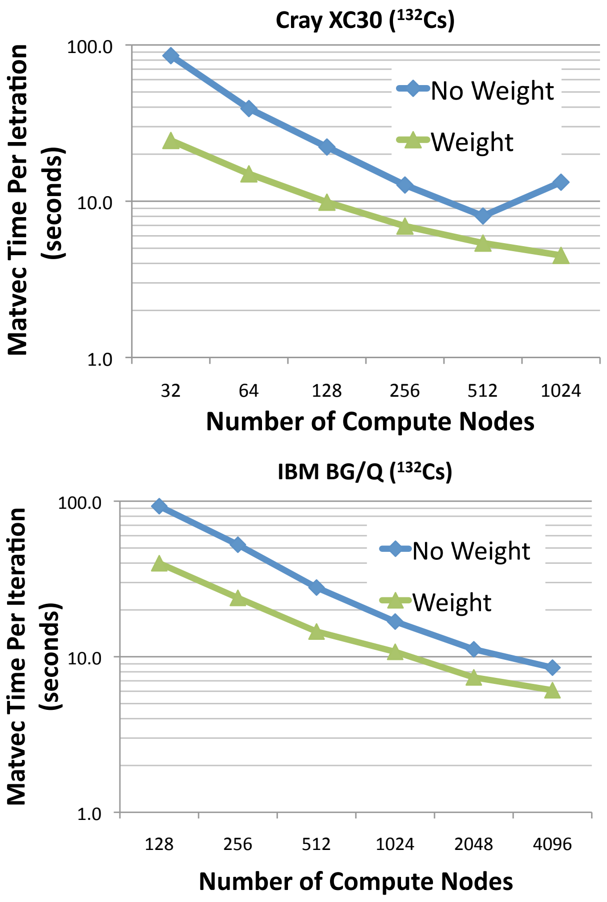

Figure 4: Matrix-vector execution time per iteration with and without our empirically-determined operation weights on the Cray XC30 and the IBM BG/Q for ^{132}Cs. The execution times have been improved by 30-70%.

Based on our performance observations for the execution time by operation type, we constructed a new series of empirically driven weights (Table 2) for both the XC30 and the BG/Q running either the ^{132}Cs or the ^{112}Xe problems. The weights need to be adjusted accordingly if different number of OpenMP threads per MPI process are used. In effect, one may profile performance using a limited number of Lanczos iterations, limited scale, or a suitably general problem, populate the weights, then run the full solver.

Using the new weights, Figure 4 highlights the benefit to matvec execution time from improved load balance on the ^{132}Cs problem on the XC30 and BG/Q. We see that matvec performance has been improved by 30-70% at all concurrencies with similar benefits for ^{112}Xe (not shown). We also develop a simple metric to measure the load imbalance of the matvec times across processes. The metric is computed as a ratio between maximum and average time. Higher values indicate worse load imbalance across processes. Initially, the ratio is between 4.3-5.1 and 3.5-5.0 for all concurrency configurations for the ^{132}Cs problem on the XC30 and BG/Q, respectively. By applying weights, the ratios are reduced to 1.3-1.8 and 1.4-3.0. There remains significant room for further improvement.

Although MPI time (reductions and broadcasts) remains roughly constant across scales, the reduction in local computation in this strong scaling regime has resulted in MPI time exceeding 55% of the total matvec time. When MPI dominates, scalability will be inhibited.

8. HYBRID PROGRAMMING EFFECT

Multi- and manycore continue to dominate the HPC landscape. At one extreme, one may run one process per core. Although this obviates thread parallelization, this approach puts tremendous pressure on the communication library and network, and in the context of BIGSTICK, duplicates vast amounts of data. At the other extreme, one may run one process per node. Unfortunately, this approach can under utilize the network (a single core might not be capable of saturating the network) and requires extremely efficient thread parallelization on potentially NUMA architectures. In reality, it is imperative to tune the MPI vs. OpenMP parallelization to determine the application- and architecture-specific optimal balance.

Figure 5 presents Lanczos iteration execution time on the Cray XC30 and IBM BG/Q platforms as a function of node concurrency and hybrid programming model configuration. For a given node concurrency, the hardware resources used are fixed across all hybrid programming model configurations. That is, on the Cray XC30, the product of MPI processes per node and OMP threads per process is always 24 while on the IBM BG/Q, the product is always 64. OpenMP is implemented inside the processing of opbundles. It is easy to partition the work to avoid write conflicts within an opbundle.

We observe that matrix-vector multiplication (Matvec) time is relatively insensitive to the choice of threading. This is true even on the Cray XC30 despite the threat of NUMA effects when running a single process per node which usually results in a factor of 2× loss in performance without proper optimization. We attribute this result partly to the presorting of the data array so that each thread will mostly only access its own part of the data and partly to the heavy local computations to generate the matrix element. An advantage of higher OpenMP parallelism is that the table memory for the generation of matrix elements is shared. Computation time being roughly equal would then favor more threads per process.

For reorthogonalization we observe that execution times (labeled as Reorth) are clearly sensitive to the number of processes per node particularly on the IBM BG/Q platforms. The clear result is that running one MPI process per node (24 OpenMP threads on the Cray XC30 and 64 OpenMP threads on the IBM BG/Q) achieves the best reorth performance. The cause is related to BIGSTICK's code design. When fewer MPI processes are used, the number of processes in the subcommunicators tend to be smaller, but the volume of communication does not change. Such configurations reward reduced process concurrency by replacing explicit messaging with cache-coherent communication. To be clear, although many MPI libraries are optimized with shared memory implementations, threaded environments both avoid duplication of data by using a single copy of shared tables in cache-coherent shared memory, and avoid DRAM bandwidth-expensive broadcasts and reductions on node.

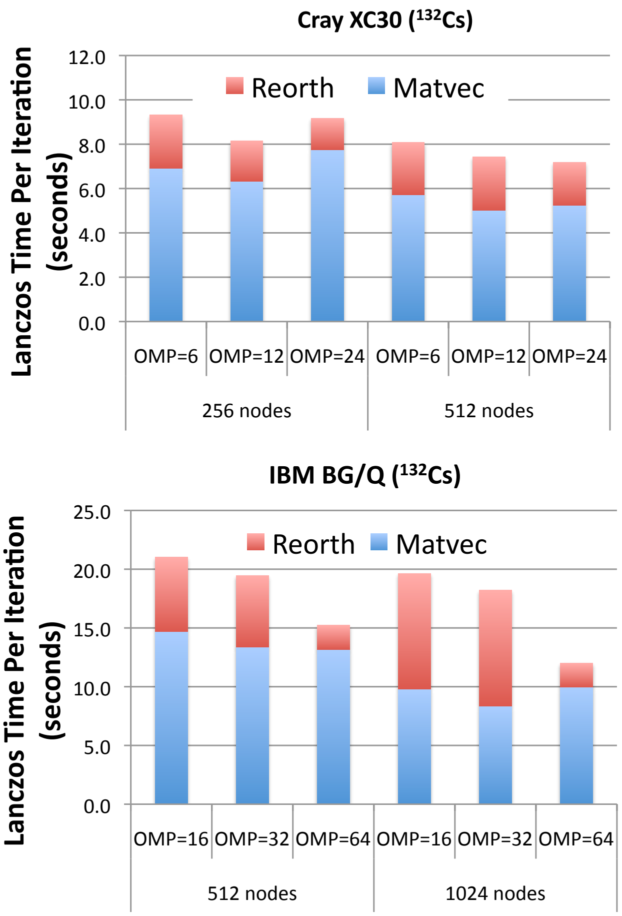

Figure 5: The hybrid programming effect. On the Cray XC30, the product of MPI processes per node and OMP threads per process is always 24 while on the IBM BG/Q, the product is always 64.

Overall, we found that the best performance is obtained on the IBM BG/Q when we use one MPI process of 64 OpenMP threads per node. On the Cray XC30, the performance differences are more subtle and dependent on node concurrency. The best result is the combinational effect of both the matvec and reorth phases and the performance difference across different combinations is relatively small.

9. REORTHOGONALIZATION

In order to maintain the mutual orthogonality of the Lanczos vectors, reorthogonalization is an essential step to correct the numerical roundoff errors. As shown in Figure 3, the reorthogonalization process will become the performance scaling bottleneck at concurrencies of about 1K nodes. In this section, we investigate three approaches improving the performance and scalability of reorthogonalization.

9.1 Process Rank Reordering

As described earlier in Section 3, both the scatter and gather collective operations are performed inside the *reorth* subcommunicator. In the current implementation, the member processes are assigned in a manner akin to round-robin in order to simultaneously balance the workload and storage requirements of reorthogonalization.

One alternative is to assign processes with contiguous world ranks to form the reorth communicator group. The rational behind is that processes with contiguous logical ranks have a better chance to be automatically mapped to the same or neighboring physical nodes and achieve better performance. Figure 6 displays the times spent on the reorthogonalization phase, labeled "Contiguous". The times for the implementation where MPI+OpenMP has been tuned are labeled as "Tuned MPI+OpenMP". The results are collected with the best combination of MPI and OpenMP threads as discussed in Section 8, i.e., running 1 MPI process with 64 OpenMP threads on each IBM BG/Q node and 2 MPI processes with 12 OpenMP threads on each Cray XC30 node.

On the Cray XC30 platform, contiguous assignment delivers similar performance with the tuned MPI+OpenMP implementation, aside from the one data point for ^{112}Xe (1K nodes), where reorthogonalization time jumps unexpectedly. The MPI profiling shows that this sudden jump is caused by the gather operation. Recent studies [18, 7] on the dragonfly network find that contiguous assignment may not be the best strategy. Our performance results are consistent with such findings. Conversely, on the IBM BG/Q platform, using a contiguous assignment of ranks can deliver similar or better performance for both the ^{132}Cs and ^{112}Xe data sets. This suggests optimization of the reorthogonalization step will be platform-dependent.

9.2 Fusing Collectives

Nominally, matvec requires a MPI_reduce while reorthogonalization requires a MPI_scatter. One is motivated to combine these collective operations via MPI_reduce_scatter and thereby defer the reduction. The advantage is that we can perform the reduction operation directly on the destination process. Based on the simple, non-contention network models used to analyze the MPI collective performance [31], the time taken by MPI_Reduce_scatter is

$$(P-1)\alpha + \frac{P-1}{P}n\beta + \frac{P-1}{P}n\gamma$$

for long messages using pair-wise exchange algorithm, where P is the number of MPI processes, α is the latency per message, β is transfer time per bytes, n is the message size in bytes, and γ is computation cost per byte to perform the local reduction operation. The time taken by MPI_Reduce is

$$2\log(P)\alpha + 2\frac{P-1}{P}n\beta + \frac{P-1}{P}n\gamma$$

while for MPI_scatter, the time is

$$\log(P)\alpha + \frac{P-1}{P}n\beta$$

The performance of BIGSTICK is most sensitive to the bandwidth terms due to its large collective message sizes. If we compare the bandwidth terms of MPI_Reduce_scatter with the sum of MPI_Reduce and MPI_scatter, the former should improve the performance substantially.

Unfortunately, the MPI_reduce_scatter can not be directly applied here. First, different communicators are used for the reduce and gather operations in BIGSTICK. Secondly, the MPI function MPI_reduce_scatter can only scatter the data based on the process rank in the communicator. It does not support the displacement parameter as in the MPI_Scatter function to scatter the data based on offsets. The first problem (different communicators) can be addressed by duplicating the *reorth* communicator from the *row* communicator. The potential drawback is that the Lanczos vectors will not be evenly distributed among all processes. But the difference should be small and tolerable. To address the second problem (displacement parameter), we developed our own implementation using the pairwise algorithms used by MPICH [31]. This algorithm requires $P-1$ steps. At step s, process with rank r will send data to process $mod(r+s,p)$ and receive data from process $mod(r-s+p,p)$. The communicated data is only the data needed for the scattered result on the process. After process r has received the data, a reduction operation will be performed locally with OpenMP enabled.

Using this new scheme, reorthogonalization execution times are shown in Figure 6 labeled with "Fusing". Significant performance improvement has been observed on the Cray XC30. Compared with the tuned implementation, the reorthogonalization times have been reduced over 50%. Moreover, the time spent in reductions in the matrix-vector multiplication phase has also been reduced by over 5×. Figure 7 displays the reduction times on the Cray XC30 platform for the original and fusing algorithms.

On the IBM BG/Q platform, in most cases, we can observe the explicit advantage of fusing the two collective operations. However, the performance benefits are much smaller compared with the Cray XC30 platform, especially for the ^{132}Cs data set when larger number of nodes are used. This can likely be explained by the performance capabilities of these two network architectures when performing large collective operations. The more sensitive XC30 sees a large speedup when collectives are fused and optimized.

9.3 Judicious Parallelization

As one moves to ever higher concurrencies, the matrix-vector multiplication time generally scales well. However, the reorthogonalization time saturates, or more than likely, increases substantially. At sufficiently high concurrency, reorthogonalization becomes a bottleneck to overall performance.

As this is a strong scaling regime, we choose to forestall this bottleneck by running reorthogonalization at reduced concurrency. That is, we will perform the matvec with the full number of MPI processes, but judiciously choose the number of MPI processes for the reothogonalizration in order to maximize its performance. In extreme case, we may use only 1 MPI process in each reorth communicator. Although this will substantially reduce the scatter and gather time (fewer communicating partners), the size of the Lanczos vectors stored on the MPI processes involved in the reorthogonalization will increase correspondingly (as well the local time to perform the dot products). Ideally, we expect this algorithm to deliver constant time (or at least proportional to the dot product time). At high concurrencies, this algorithm will outperform the other algorithms.

Figure 6 displays the times for this approach labeled as *Judicious*. On the IBM BG/Q platform, the reorthogonalization times remains nearly constant with increasing con-

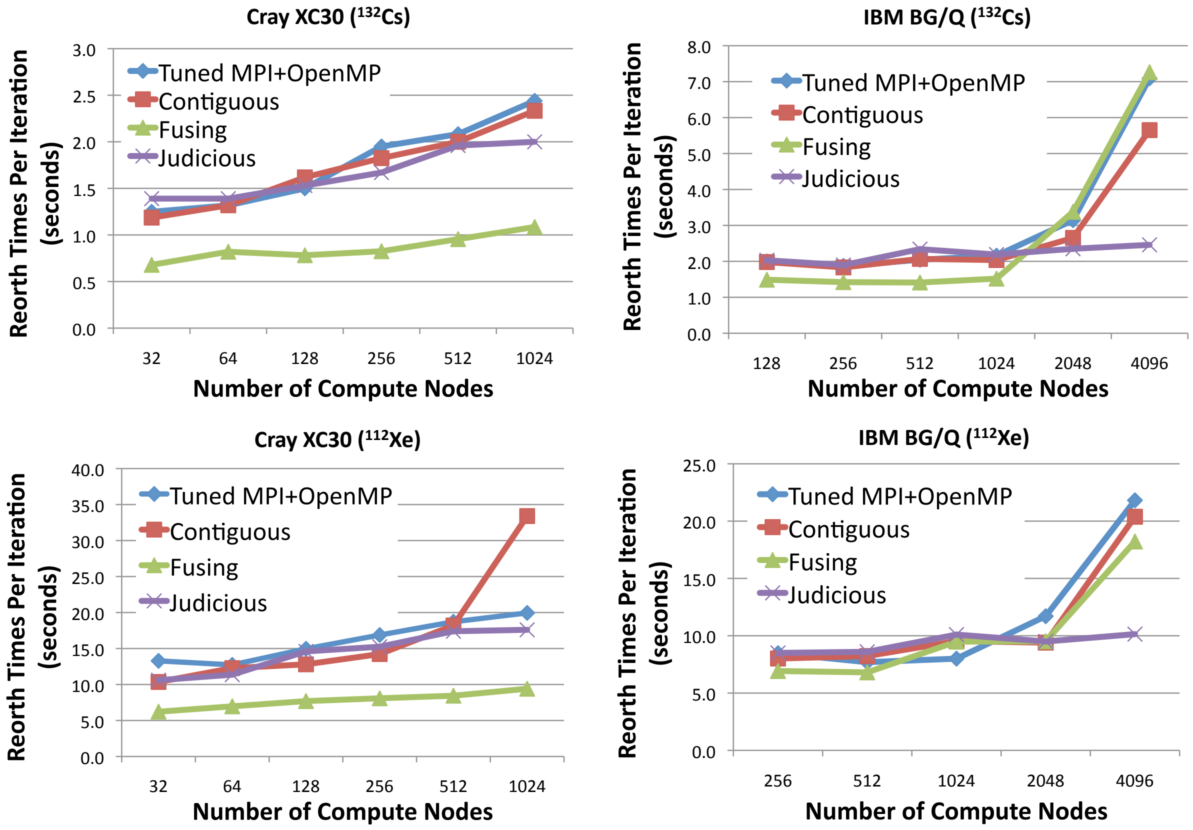

Figure 6: Reorthogonalization execution time per iteration on the Cray XC30 and the IBM BG/Q for ^{132}Cs and ^{112}Xe for each algorithm. Observe the platform- and scale-dependent optimal implementation.

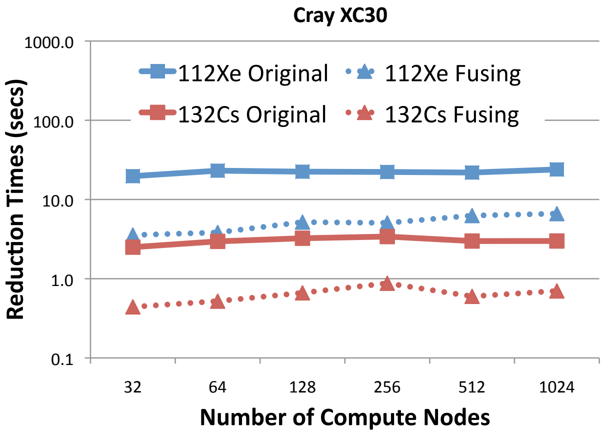

Figure 7: Time spent in matvec's MPI reduction on the Cray XC30 platform when using either the original or collective fusing reorthogonalization algorithms.

currency. Below 1K nodes, our optimized reorthogonalization algorithms are not differentiated. However, at extreme scale, this approach is 2-3× faster. Conversely, on the Cray XC30, while the reduced parallelization scheme is faster than the original implementation with tuned MPI+OpenMP at 1K nodes, it is still 2× slower than using the custom fusing approach. Platform-specific (network topology-specific) algorithms and parallelization schemes are clearly a necessity, yet an undesirable solution.

10. SCIENCE RESULTS

Nuclei are important laboratories for exploring new physics, such as dark matter detection, and BIGSTICK and other nuclear structure codes enable interpretation of those experiments. At the end of a BIGSTICK run the wave functions for the low lying states can be written to disk and post-processed to extract cross-sections, decay rates, etc. In the case of dark matter, we do not know the strength of the coupling between dark matter particles and nucleons, nor even its character. By conducting experiments with a variety of different targets we can either determine the coupling or establish upper limits. As an example, we present in Table 3 the spin response defined in equation (2) for several stable isotopes used in ongoing dark matter detection experiments; we choose nuclei with an odd number of nucleons because they have the largest spin response. Calculations for other nuclei and other couplings are in progress.

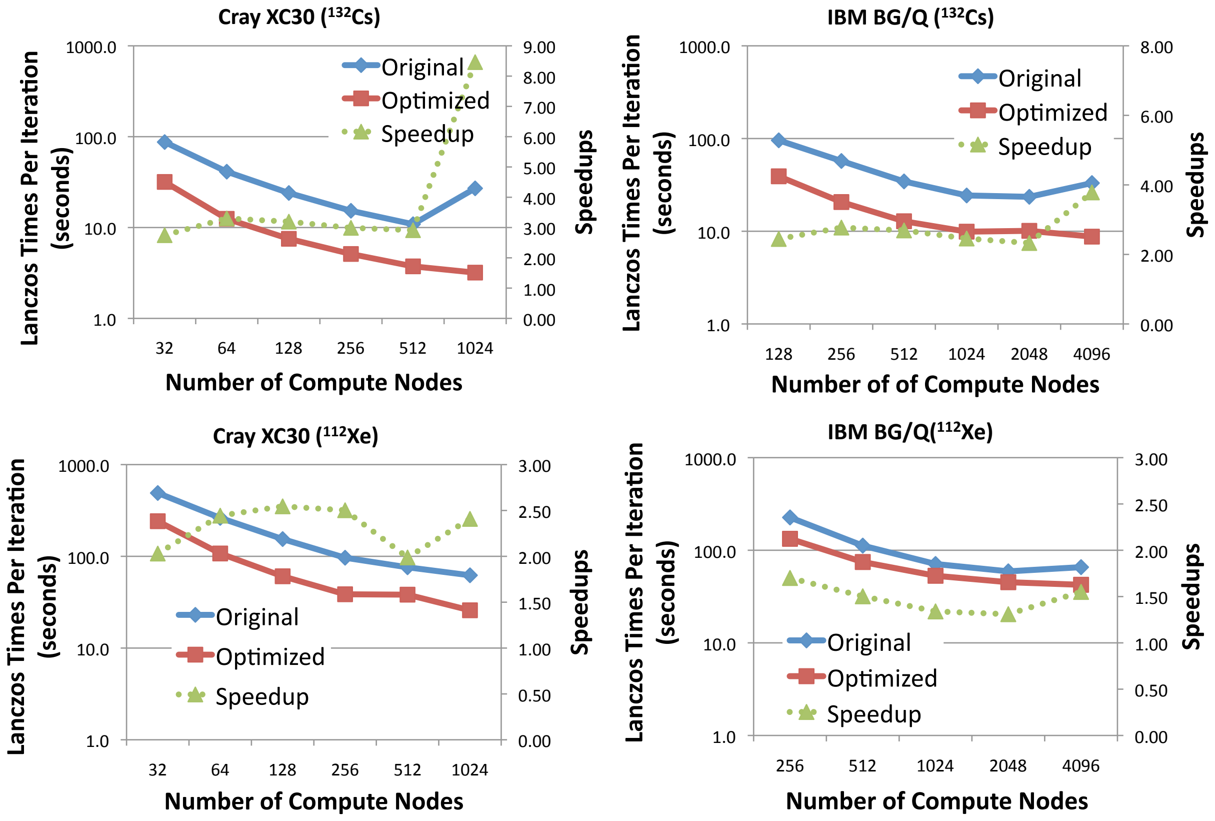

Figure 8: Lanczos iteration time before and after optimization for ^{132}Cs and ^{112}Xe on the Cray XC30 and the IBM BG/Q platforms. Observe substantial speedups at scale for the ^{132}Cs problem.

Table 3: Isoscalar and Isovector Spin Response

Isotope	Spin Response		Basis
	Isoscalar	Isovector	Dimension
^{127}I	0.7087	0.5011	1.3 Billion
^{133}Cs	0.8404	0.7235	0.2 Billion
^{131}Xe	0.5251	0.4627	0.2 Billion
^{129}Xe	0.0005	0.0005	3.1 Billion

11. SUMMARY AND FUTURE WORK

In this paper, we examined the performance and scalability of BIGSTICK, a matrix-free configuration interaction code. In order to address the observed performance and scalability bottlenecks, we implemented several optimizations including empirically-driven load balancing, tuning of the process vs. thread parallelization, fusing MPI collective operations, and running some operations at reduced concurrency. Figure 8 presents the performance of our ultimate optimized version compared to the original implementation. For clarity, we add a third curve to denote the attained speedup of optimized over original.

On both machines, we eliminate the scaling inflection point for ^{132}Cs at extreme scale to ensure monotonic strong scaling. On the IBM BG/Q platform, we attain a speedup of about 2.5-4× for ^{132}Cs and about 1.5× for ^{112}Xe (a problem size less sensitive to load imbalance). The best performance is obtained using the fusing algorithm up to 1K nodes. Beyond that point, the judicious (reduced) parallelization of reorthogonalization attains the best performance. On the Cray XC30 platform, the best performance at all concurrencies is obtained by the fusing algorithm. Collectively, optimization provide a 3-8× speedup for the ^{132}Cs problem and up to 2.5× for ^{112}Xe (the differences in scalability and parallel efficiency for ^{132}Cs and ^{112}Xe are attributable to the differences in their sizes and densities). The IBM BG/Q's scaling is substantially impaired beyond 4K nodes as it has exhausted most of the available coarse-grained (opbundle) parallelism. Based on these insights we believe one could probably scale the Cray XC30 to perhaps 2K or 4K nodes. Nevertheless, we observe that the Cray XC30 at 1K nodes is up to 2.6× faster than the IBM BG/Q at 4K nodes for the ^{132}Cs problem.

Stored matrix representations like MFDn will require 1-2 orders of magnitude more memory than BIGSTICK to describe a system. Whereas BIGSTICK can be sensitive to load balancing challenges as the cost of an opbundle can differ by more than 7× within a platform and by more than 10× across platforms, highly-optimized, distributed SpMV implementations are easier to load balance and scale. BIGSTICK is very attractive as it can solve large systems with very few nodes and strong scale to 1000s of nodes. However, at extreme scale, where network and reorthogonalization times dominate, the more easily optimized stored matrix codes will likely be faster. Nevertheless, BIGSTICK's memory-efficient representation will allow for solutions to extreme-scale systems that stored matrix representations cannot describe.

To further enhance performance and scalability of BIGSTICK, we will combine our collective fusion and judicious parallelization techniques. In addition, we will explore alternate mappings of process ranks to fragments in order to mitigate any tapering in the network and finite bisection bandwidth.

Finally, as processors like the Knights Landing Xeon Phi move towards hierarchical memory architectures, the capability and performance of stored matrix approach will be increasingly challenged as one cannot store large matrices in the "fast" memory. We believe the factorization approach used in BIGSTICK is more amenable to such architectures as one can fit a representation of the matrix in fast memory. Moreover, as we have demonstrated, BIGSTICK delivers high performance up to 64 threads. We thus believe it will efficiently implement the matvec by meeting the on-chip concurrency demands of the KNL processor. We will thus move BIGSTICK to the KNL-based Cori at NERSC when available in order to quantify the impacts of hierarchical memory and substantially higher node and core concurrency.

Acknowledgements

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231 (Shan and Williams), and by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics, under Award Numbers DE-AC02-05CH11231 (McElvain), DE-FG02-96ER40985 (Johnson), and Contract No. DE-AC52-07NA27344 and Work Proposal No. SCW0498 (Ormand). This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and the Argonne Leadership Computing Facility, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

12. REFERENCES

[1] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary. Improving the scalability of a symmetric iterative eigensolver for multi-core platforms. *Concurrency and Computation: Practice and Experience*, 26:2631–2651, 2014.

[2] E. Aprile and T. Doke. Liquid xenon detectors for particle physics and astrophysics. *Rev. Mod. Phys.*, 82:2053–2097, Jul 2010.

[3] B. Brown, A. Etchegoyen, and W. Rae. Computer code OXBASH: the Oxford University-Buenos Aires-MSU shell model code. *Michigan State University Cyclotron Laboratory Report No. 524*, 1985.

[4] B. A. Brown and W. D. M. Rae. The Shell-Model Code NuShellX@MSU. *Nuclear Data Sheets*, 120:115–118, 2014.

[5] B. A. Brown and B. H. Wildentha. Status of the nuclear shell model. *Annual Review of Nuclear and Particle Science*, 38:29–66, 1988.

[6] P. Brussard and P. Glaudemans. *Shell-model applications in nuclear spectroscopy.* North-Holland Publishing Company, Amsterdam, 1977.

[7] R. D. Budiardja, L. Crosby, and H. You. Effect of Rank Placement on Cray XC30 Communication Cost. *The Cray User Group Meeting*, 2013.

[8] E. Caurier, G. Martinez-Pinedo, F. Nowacki, A. Poves, J. Retamosa, and A. P. Zuker. Full $0\hbar\omega$ shell model calculation of the binding energies of the 1f7/2 nuclei. *Phys. Rev. C*, 59:2033–2039, 1999.

[9] E. Caurier, G. Martinez-Pinedo, F. Nowacki, A. Poves, and A. P. Zuker. The shell model as a unified view of nuclear structure. *Reviews of Modern Physics*, 77:427–488, 2005.

[10] E. Caurier and F. Nowacki. Present status of shell model techniques. *Scopus Preview*, 30:705–714, 1999.

[11] C. J. Christopher. *Essentials of Computational Chemistry*, pages 191–232. John Wiley & Sons, Ltd., ISBN 0-471-48552-7, 2002.

[12] www.nersc.gov/systems/edison-cray-xc30/.

[13] A. R. Edmonds. *Angular momentum in quantum mechanics*. Princeton University Press, 1996.

[14] J. Ellis, A. Ferstl, and K. A. Olive. Constraints from accelerator experiments on the elastic scattering of CMSSM dark matter. *Physics Letters B*, 532(3-4):318 – 328, 2002.

[15] J. Engel, S. Pittel, and P. Vogel. Nuclear physics of dark matter detection. *International Journal of Modern Physics E*, 1(01):1–37, 1992.

[16] M. W. Goodman and E. Witten. Detectability of certain dark-matter candidates. *Phys. Rev. D*, 31:3059–3063, Jun 1985.

[17] P. Hamilton, M. Jaffe, J. M. Brown, L. Maisenbacher, B. Estey, and H. Muller. Viewpoint: More power to atom interferometry. *Physics Review Letters*, 114:100405, 2015.

[18] N. Jain, A. Bhatele, N. J. W. Xiang Ni, and L. V. Kale. Maximizing Throughput on a Dragonfly Network. *The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[19] C. W. Johnson, W. E. Ormand, and P. G. Krastev. Factorization in large-scale many-body calculations. *Computer Physics Communications*, 184:2761–2774, 2013.

[20] G. Jungman, M. Kamionkowski, and K. Griest. Supersymmetric dark matter. *Physics Reports*, 267(5-6):195 – 373, 1996.

[21] K. Kandalla, H. Subramoni, A. Vishnu, and D. K. D. Pandaj. Designing Topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters: Case Studies with Scatter and Gather. *IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[22] S. Kumar, A. Mamidala, P. Heidelberger, D. Chen, and D. Faraj. Optimization of MPI collective operations on the IBM Blue Gene/Q supercomputer.

International Journal of High Performance Computing Applications, 28:450–464, 2014.

[23] D. A. B. Miller. *Quantum Mechanics for Scientists and Engineers*. Cambridge University Press, ISBN-13: 978-0521897839, 2008.

[24] http: //www.alcf.anl.gov/user-guides/mira-cetus-vesta/.

[25] https://www.jyu.fi/fysiikka/en/research/accelerator/ nuctheory/Research/Shellmodel.

[26] H. Potter, D. Oryspayev, P. Maris, M. Sosonkina, and et. al. Accelerating Ab Initio Nuclear Physics Calculations with GPUs. *Proc. 'Nuclear Theory in the Supercomputing Era - 2013' (NTSE-2013)*, 2014.

[27] I. Shavitt. The history and evolution of configuration interaction. *Molecular Physics*, 94:3–17, 1998.

[28] P. Sternberg, E. Ng, C. Yang, P. Maris, J. Vary, M. Sosonkina, and H. V. Le. Accelerating configuration interaction calculations for nuclear structure. *The Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[29] www.cs.virginia.edu/stream/ref.html.

[30] L. E. Strigari. Galactic searches for dark matter. *Physics Reports*, 531(1):1 – 88, 2013. Galactic searches for dark matter.

[31] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH. *10th European PVM/MPI User's Group Meeting*, 2003.

[32] J. Toivanen. Efficient matrix-vector products for large-scale nuclear shell-model calculations. *http://inspirehep.net/record/728378/*, 2006.

[33] R. R. Whitehead, A. Watt, B. J. Cole, and I. Morrison. Computational Methods for Shell-Model Calculations. *Adv. Nuclear Phys. Vol. 9, pp 123-176*, 1977.