Recent Advances in MFDn

Chao Yang[a], Hasan Metin Aktulga[a], Pieter Maris[b], Esmond Ng[a] and James Vary[b]

[a] *Computational Research Division, Lawrence Berkeley National Lab, Berkeley, CA 94720*
[b] *Department of Physics and Astronomy, Iowa State University, Ames, IA 50011*

Abstract

The Many-body Fermion Dynamics for nuclear physics (MFDn) code was originally developed by James Vary and his colleagues for performing nuclear configuration interaction (CI) calculations. We describe a number of recent algorithmic and implementation advances in MFDn that enabled it to achieve high performance, and allowed scientists to study properties of light nuclei with high accuracy on modern high performance computers.

Keywords: *High-performance computing; configuration interaction method; matrix diagonalization*

1 Introduction

The MFDn (Many Fermion Dynamics for nuclear structure) software was developed by James Vary and his collaborators at Iowa State University [1, 2]. It is a computational tool for studying nuclear structure.

In MFDn, the nuclear Hamiltonian is evaluated in a large harmonic oscillator single-particle basis and diagonalized by iterative techniques to obtain the low-lying eigenvalues and eigenvectors. The eigenvectors are then used to evaluate a suite of experimental quantities to test accuracy and convergence issues. In several respects, the approach is similar to the Full Configuration Interaction (FCI) method in other fields [3, 4]. We often obtain convergence, either by direct diagonalization or simple extrapolation, and we then claim we have the result of an exact calculation.

In this paper, we describe a number of recent algorithmic and implementation advances that made MFDn highly efficient on modern high performance computers.

The paper is organized as follows. In Section 2, we will review the general formulation of the nuclear many-body problem and the nuclear CI methodology used in MFDn for computing a few lowest energy states of a nuclear structure. The implementation details of several key components of MFDn are presented in Sections 3 to 7.

2 Background

The structure of an atomic nucleus with k nucleons is described by a many-body wavefunction $\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_k)$, where $\mathbf{r}_j \in \mathbb{R}^3$, $j = 1, 2, \ldots, k$. The wavefunction satisfies the many-body Schrödinger equation

$$H\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_k) = \lambda\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_k), \tag{1}$$

Proceedings of International Conference 'Nuclear Theory in the Supercomputing Era — 2013' (NTSE-2013), Ames, IA, USA, May 13–17, 2013. Eds. A. M. Shirokov and A. I. Mazur. Pacific National University, Khabarovsk, Russia, 2014, p. 272.
http://www.ntse-2013.khb.ru/Proc/Yang.pdf.

where H is a many-body Hamiltonian that relates a nucleus configuration defined by Ψ to the energy of the system. We denote the energy of the system by λ. The many-body Hamiltonian H is defined as

$$H = \frac{1}{k} \sum_{i<j} \frac{(\mathbf{p}_i - \mathbf{p}_j)^2}{2m} + \sum_{i<j=1}^{k} V_n(\mathbf{r}_i - \mathbf{r}_j), \tag{2}$$

where m is the nucleon mass, $\mathbf{p}_i$ is a momentum operator, and $V_n(\mathbf{r}_i - \mathbf{r}_j)$ is a two-body potential operator that describes the interaction between the ith and jth nucleons. A more accurate treatment of the problem may include three-body potentials. Clearly, the wavefunction Ψ is an eigenfunction of H associated with the eigenvalue λ.

For nuclei that consist of a few nucleons (less than five), there are several methods to solve (1) directly. However, as k becomes larger, the size of the problem will become so large that approximate methods are necessary. One way to overcome the dimensionality explosion is to project the many-body Hamiltonian into a lower dimensional subspace $\mathcal{S}$ spanned by a set of *Slater determinants* defined as

$$\Phi_a(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_k) = \frac{1}{\sqrt{k!}} \begin{vmatrix} \phi_{a_1}(\mathbf{r}_1) & \phi_{a_2}(\mathbf{r}_1) & \dots & \phi_{a_k}(\mathbf{r}_1) \\ \phi_{a_1}(\mathbf{r}_2) & \phi_{a_2}(\mathbf{r}_2) & \dots & \phi_{a_k}(\mathbf{r}_2) \\ \vdots & \vdots & & \vdots \\ \phi_{a_1}(\mathbf{r}_k) & \phi_{a_2}(\mathbf{r}_k) & \dots & \phi_{a_k}(\mathbf{r}_k) \end{vmatrix}, \tag{3}$$

where ϕ_{a_i} is the eigenfunction associated with the a_i-th eigenvalue of a (single-particle) Hamiltonian $h = \mathbf{p}^2/2m + v(\mathbf{r})$. Conventionally, one uses a harmonic oscillator potential, which is quadratic in $\mathbf{r}$. The use of Slater determinants is a standard technique employed in quantum mechanics.

In this paper, we define the index of $\Phi_a(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_k)$ by a strictly increasing k-tuple of integers, i. e. $a = (a_1, a_2, \dots, a_k)$, where a_i is simply the index of the single-particle eigenfunction that appears in the ith column of the Slater determinant. We will refer to $\Phi_a(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_k)$ or simply a as a *many-body basis state.* We will call each component of a a *single-particle state.* Not all many-body basis states are valid because each has to satisfy a set of conditions to be defined later. We denote the set of all valid many-body basis states $\{a\}$ by $\mathcal{A}$. The size of $\mathcal{A}$ will be denoted by $n = |\mathcal{A}|$.

Suppose the desired many-body wavefunction can be well represented by a linear combination of the basis functions Φ_a $(a \in \mathcal{A})$, i. e.,

$$\Psi = \sum_{a \in \mathcal{A}} c_a \Phi_a, \tag{4}$$

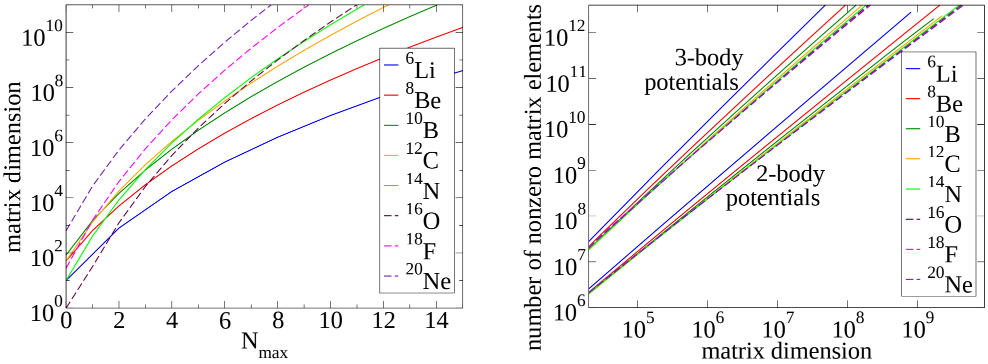
where $c_a \in \mathbb{R}$, we can then solve (1) by computing eigenpairs of a projected Hamiltonian $\hat{H}$, where

$$\hat{H}_{a,b} = \int_{\Omega} (\Phi_a^* H \Phi_b)\, d\mathbf{r}_1 d\mathbf{r}_2 \dots d\mathbf{r}_k. \tag{5}$$

Because H is self-adjoint, $\hat{H}$ is real symmetric. The eigenvector of $\hat{H}$ associated with the desired eigenvalue (energy) gives the coefficients c_a in (4).

Clearly, the dimension of $\hat{H}$, which is the total number of valid many-body basis states $|\mathcal{A}|$, depends on the total number of nucleons (k) contained in the nucleus of interest and the largest single-particle state ($a_{\max}$) allowed in $\Phi_a(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_k)$, which is implicitly determined by a constraint imposed on the total oscillator quanta ($N_{\max}$). For a large nucleus with many nucleons and large $a_{\max}$ value, n can be extremely large. However, the number of nonzero elements in $\hat{H}$ is typically very small, as we will show below.

It follows from the mutual orthogonality of all single-particle eigenfunctions ϕ_ℓ $(\ell = 1, 2, \dots, a_{\max})$ that a one-body integral in (5) becomes zero when a and b differ



(a) The growth of the matrix dimension ($|\mathcal{A}|$) with respect to $N_{\max}$

(b) The growth of number of nonzero matrix elements in $\hat{H}$ with respect to $|\mathcal{A}|$

Figure 1: The characteristics of the CI projected Hamiltonian $\hat{H}$ for a variety of nuclei.

by more than one single-particle state, and a two-body integral becomes zero when a and b differ by more than two single-particle states, etc. This observation allows us to determine many of the zero entries of $\hat{H}$ without evaluating the numerical integral in (5).

Empirical evidence suggests that the probability of two randomly chosen but valid many-body basis states sharing more than $k-2$ single-particle states is relatively low. As a result, $\hat{H}$ is extremely sparse. Figure 1 shows both the growth of the matrix dimension ($|\mathcal{A}|$) with respect to $N_{\max}$ and the growth of the number of nonzero elements in $\hat{H}$ with respect to $|\mathcal{A}|$ for a variety of nuclei for both two-body and two-plus three-body potentials. In practice, we observe that the number of non-zeros in $\hat{H}$ is proportional to $|\mathcal{A}|^{3/2}$.

To compute the eigenvalues of $\hat{H}$ efficiently on a high performance parallel computer, the following three issues must be addressed carefully:

1. The generation and distribution of the many-body basis states — This step essentially determines how the matrix Hamiltonian $\hat{H}$ or $\hat{H}_Z$ is partitioned and distributed in subsequent calculations.

2. The construction of the sparse matrix Hamiltonian $\hat{H}$ — This step is performed simultaneously on all processors. Each processor will construct its portion of $\hat{H}$ defined by the many-body basis states assigned to it. Because the positions of the nonzero elements of the Hamiltonian is not known a priori, the key to achieving good performance during this step is to quickly identify the locations of these elements without evaluating them numerically first.

3. The calculation of the eigenvalues and eigenvectors using the Lanczos iteration — The major cost of the Lanczos iteration is the computation required to perform sparse matrix-vector multiplications of the form $y \leftarrow \hat{H}x$, where x, y are both vectors. Performing efficient orthogonalizations of the Lanczos basis vectors is also an important issue to consider.

3 Parallel basis generation

Because the rows and columns of $\hat{H}$ are indexed by valid many-body basis states, the first step of the nuclear CI calculation is to generate these states so that they can be used to construct and manipulate matrix elements of $\hat{H}$ in subsequent calculations. It is desirable to generate the valid many-body basis states in parallel on different processors to 1) reduce the basis generation time, 2) allow Hamiltonian to be distributed and constructed in parallel in subsequent computation.

Because the set of valid many-body states cannot be described by a simple expression, the strategy we adopt is to enumerate all possible many-body basis states in some order and pick out the ones that satisfy a set of constraints to be defined below.

When single-particle wave functions ϕ's in (3) are chosen to be eigenfunctions of a harmonic oscillator, each a_i corresponds to a set of quantum numbers (n, l, j, m_j), where $n \geq 0$ is the quantum number that is associated with the radial component of the eigenfunction, $l \geq 0$ is the angular momentum of the single-particle, j, which is either $|l - 1/2|$ or $l + 1/2$, is the coupled spin-angular momentum, and m_j, which can assume the values of $-j, j+1, \ldots, j-1, j$ is the projection of the spin-angular momentum to a particular spatial axis, often chosen to be the z axis. Single-particle states are typically ordered by their energy levels (i. e., the corresponding eigenvalues of the harmonic oscillator Hamiltonian). The energy of the single particle associated with (n, l, j, m_j) can be labeled by $N = 2n + l$, which is degenerate. Single-particle states associated with each degenerate energy level are typically ordered by their m_j values.

If the maximum index of the allowed single-particle state is $a_{\max}$, the total number of different $\Phi_a(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_k)$ is $\begin{pmatrix} a_{\max} \\ k \end{pmatrix}$, which can be extremely large. However, in many cases, interesting physics of a nucleus can be ascertained from a much smaller model (or configuration) space that contains far fewer $\Phi_a(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_k)$'s that satisfy additional constraints. In MFDn, these constraints include

1. Oscillator excitation quanta constraint: $\sum_{i=1}^{k} N(a_i) \leq N_{\max}$, where $N(a_i)$ is the oscillator quanta with the ith single-particle, and $N_{\max}$ is known as the maximum oscillator excitation quanta chosen in advance. Clearly, the larger the $N_{\max}$ value, the larger the model space is.

2. Magnetic projection constraint: $\sum_{i=1}^{k} m_j(a_i) = m$, where $m_j(a_i)$ is the m_j value associated with the ith single-particle and m is a total magnetic projection constant chosen in advance

3. Parity constraint.

After imposing these constraints, a majority of the enumerated many-body basis states can be eliminated. A many-body basis state satisfying all three conditions above is a *valid* state. The easiest way to generate all many-body basis states is to enumerate them in a lexicographical order defined below. A many-body basis state $a = (a_1, a_2, \ldots, a_k)$ is said to be *lexicographically less than* another many-body basis state $b = (b_1, b_2, \ldots, b_k)$ if and only if there is a j for which $a_j < b_j$ and $a_i = b_i$ for all $i < j$. For example, if $a_{max} = 9$, then $(1, 3, 4, 8)$ is succeeded by $(1, 3, 4, 9)$, which is in turn succeeded by $(1, 3, 5, 6)$, and $(1, 3, 8, 9)$ is succeeded by $(1, 4, 5, 6)$. The details on how valid many-body states are enumerated can be found in [5].

To carry out nuclear CI calculation on a distributed-memory parallel computer, the projected Hamiltonian $\hat{H}$ must be partitioned and distributed among different processors. Associated with this partition is a partition and distribution of the many-body basis states.

Because $\hat{H}$ is symmetric, we generate and store only the lower triangular part of the matrix to minimize memory usage. To create the matrix and processor mapping, we first partition $\hat{H}$ into rectangular blocks of roughly the same size and map the matrix blocks in the lower triangular part of the 2D partition to different processors. Figure 2 illustrates one way to map matrix blocks to 6 processors. We label each distributed block by a processor identification (pid) number that ranges from 1 to n_p, where n_p is the total number of processors in use. Due to the particular distribution

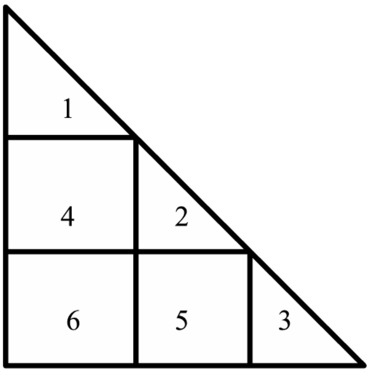


Figure 2: The projected Hamiltonian $\hat{H}$ is partitioned and distributed among 6 processors.

pattern shown in Fig. 2, the choice of n_p is not arbitrary. If we let n_d be the number of diagonal blocks in the partition, then $n_p = n_d(n_d + 1)/2$. For reasons that we will explain later n_d is typically chosen to be an odd integer. In the following, we will refer to the processors to which the diagonal blocks of $\hat{H}$ are assigned as the diagonal processors. They are labeled by 1 through n_d in Fig. 2. In MFDn, row and column communication groups are created to allow information to be passed among processors associated with row or column blocks of $\hat{H}$. As we will explain later, the matrix block to processor mapping shown in Fig. 2 is not the most efficient.

To avoid moving the matrix elements of $\hat{H}$ among different processors and to speed up the construction of $\hat{H}$, we generate the each submatrix block in (2) simultaneously. Each processor must have two sets of many-body basis states (one corresponding to the row indices and the other corresponding to the column indices except when the submatrix block is one the diagonal. In that case, the row many-body basis states are identical to the column many-body basis states.)

In MFDn, these many-body states are generated in parallel on the diagonal processors only. The ith diagonal processor is responsible for enumerating $n_d \cdot m + i$th many-body state and checking its validity. The invalid ones are simply discarded. This generation scheme naturally leads to a nearly cyclic distribution of the valid many-body states. Once a set of valid many-body basis states generated on the ith diagonal processor, they are broadcast to all processors that belong to same column and row processor groups that contain the ith diagonal processor.

The nearly cyclic distribution of the valid many-body states leads to balance load among different processors because both the sizes of the blocks assigned to different processors and the number of nonzero matrix elements of $\hat{H}$ are approximately the same [5].

Sometimes, it is convenient to partition or group valid many-body states in some way so that they can be generated one group at a time. One way to perform such a partition is to use the $\{n, l, j\}$ quantum numbers associated with all single-particle states in Φ_a as guidance. In this case, a group of many-body basis states associated with a particular sequence of (n, l, j) quantum numbers $\{\bar{n}_i, \bar{l}_i, \bar{j}_i\}$ $(i = 1, 2, ..., k)$ is defined to be

$$\mathcal{G}(\{\bar{n}_i, \bar{l}_i, \bar{j}_i\}) \equiv \{\Phi_a | n(a_i) = \bar{n}_i, l(a_i) = \bar{l}_i, j(a_i) = \bar{j}_i), \quad \text{for} \quad i = 1, 2, ..., k\}. \tag{6}$$

This particular choice of grouping is useful because the set of many-body states that belong to the same $\mathcal{G}(\{\bar{n}_i, \bar{l}_i, \bar{j}_i\})$ is invariant under the magnitude square of the total spin-angular momentum operator $\hat{\mathbf{J}}$, which is often denoted by J^2.

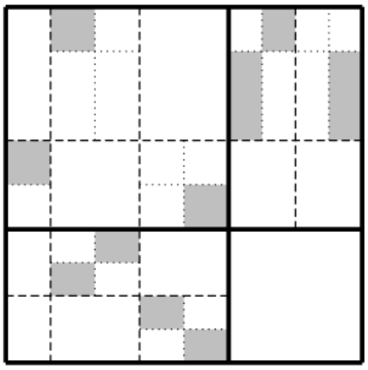


Figure 3: A three-level blocking of a portion of the Hamiltonian matrix $\hat{H}$ distributed to an off-diagonal processor. The first (coarsest) level blocks are bordered by solid lines. The second level of blocks are bordered by thinner dashed lines. The finest level blocks are bordered by dotted lines, and those blocks containing non-zeros are shaded.

4 Hamiltonian construction

A pair of many-body states that differ by no more than K single-particle states with respect to a Hamiltonian that contains at most K-body interactions is called an interacting pair. A non-interacting pair corresponds to a zero matrix element indexed by these two states. Such a matrix entry does not need to be evaluated or stored.

Once all many-body basis states have been generated, we can determine the nonzero structure of the Hamiltonian matrix by comparing each pair of many-body basis states. However, this brute-force approach of exhaustive comparison requires time proportional to the square of the number of many-body states. Even though each such pairwise test is very simple, the sheer number of them renders this process prohibitively expensive.

A more efficient way to determine the nonzero structure of the Hamiltonian was developed in [5]. It is based on the observation that the Hamiltonian matrix typically contains large blocks of zeros. The basic idea is to identify these large blocks of zeros by separating many-body basis states into different groups and assigning a group identification (id) to each group. Instead of performing pairwise comparisons of many-body basis states, we can first perform pairwise comparisons of group id's. Such comparison allows us to identify a block that contains zeros only with a single comparison. Pairwise comparisons of individual many-body basis states only need to be performed for blocks that are identified to contain nonzero matrix elements. Furthermore, this approach can be implemented recursively, which leads to a hierarchical scheme for identifying zero matrix blocks. Figure 3 gives a schematic illustration of what a three-level blocking of a Hamiltonian will look like. The shaded blocks represent the finest level blocks that contain nonzero matrix elements. In this particular case, a large block of zeros, the (2,2)-block bordered by solid lines, is identified at the coarsest level. Nine intermediate-sized zero blocks can be found at the second level. Sixteen small zero blocks can be seen at the finest level.

5 Algorithms for computing a few eigenvalues

A natural algorithm for computing a selected few eigenvalues and their corresponding eigenvectors of $\hat{H}$ is an iterative method that does not require storing all $|\mathcal{A}| \times |\mathcal{A}|$ matrix elements. In nuclear physics, the eigenvalues of interest are those at the low end of the spectrum of $\hat{H}$ because they describe the ground and the first few excited states of the nucleus. In MFDn, these eigenvalues are computed by the Lanczos method, which projects $\hat{H}$ into a *Krylov* subspace $\mathcal{K}(\hat{H}, v_0) = \mathrm{span}\{v_0, \hat{H}v_0, \ldots, \hat{H}^{\ell-1}v_0\}$ of dimension $\ell \ll n$, where $v_0 \in \mathbb{R}^n$ is an arbitrarily chosen starting vector. If $V = (v_1, v_2, \ldots, v_\ell)$ consists of an orthonormal basis of $\mathcal{K}(\hat{H}, v_0)$, the Lanczos method can be described by

$$\hat{H}V = VT + fe_\ell^T, \tag{7}$$

where $T = V^T\hat{H}V$ is an $\ell \times \ell$ tridiagonal matrix that represents the projection of $\hat{H}$ into $\mathcal{K}(\hat{H}, v_0)$, f is a residual vector that satisfies $V^Tf = 0$, and e_ℓ is the ℓ-th column of the identity matrix. Approximations to eigenvalues of $\hat{H}$ can be obtained by computing eigenvalues of the much smaller matrix T. If q is an eigenvector of T associated with the eigenvalue θ, then $z = Vq$ is the approximation to an eigenvector of $\hat{H}$.

It is well known that well separated extremal eigenvalues converge rapidly in the Lanczos iteration [6]. Convergence can be further improved by carefully choosing the starting vector v_0 and refining it using the implicitly restarted Lanczos algorithm developed in [7] and implemented in [8]. The major cost of the algorithm is the matrix-vector multiplication $w \leftarrow \hat{H}v$ required at each iteration.

An alternative way to compute the k algebraically smallest eigenvalues is to formulate the eigenvalue problem as the following constrained minimization problem

$$\min_{Z^TZ=I} \mathrm{trace}(Z^T\hat{H}Z), \tag{8}$$

where $Z \in \mathbb{R}^{|\mathcal{A}|\times k}$. This constrained minimization problem can be solved by preconditioned Davidson–Liu method [9] or the locally optimal block preconditioned conjugate gradient method [10]. Without a preconditioner, the convergence properties of these methods are similar to that of the Lanczos algorithm. However, when a good preconditioner is available, these methods can be much faster. Sparse matrix vector multiplication constitutes the major cost of this algorithm also.

Because it is not yet clear how to construct a good preconditioner for this type of problem, we will focus on the Lanczos algorithm in the subsequent discussion.

6 Scalable implementation

We now describe some techniques for implementing the Lanczos algorithm efficiently on large-scale distributed memory parallel high performance computers. The computational cost of the Lanczos iteration is dominated by the sparse matrix vector multiplications (SpMV) $w \leftarrow \hat{H}v$ required to expand the Krylov subspace, as well as dense matrix vector multiplications required to maintain orthonormality among columns of V.

6.1 Topology aware task-to-processor mapping for SpMV

To perform the SpMV operations efficiently on a lower triangular processor grid laid out as in Fig. 2, each input vector is partitioned among the diagonal processors.

A natural way to distribute a vector v to be multiplied by the distributed A matrix is to partition it conformally with the column partition of A into n_d subvectors $\{v_i\}$, $i = 1, 2, \ldots, n_d$ as shown in Fig. 4. Row and column communication groups are set up to allow v_i to be broadcast among processing units that lie on the ith row or column of the triangular grid. If we denote the submatrix of A assigned to the (i, j)th processing unit by $A_{i,j}$, each processing unit performs two SpMVs of the form $w_i = A_{i,j}v_j$ and $w_j = A_{i,j}^T v_i$. As depicted in Fig. 4, two reductions are required (one along the row communication groups and one along the column communication groups) to merge local products w_i and w_j to form the global result vector w.

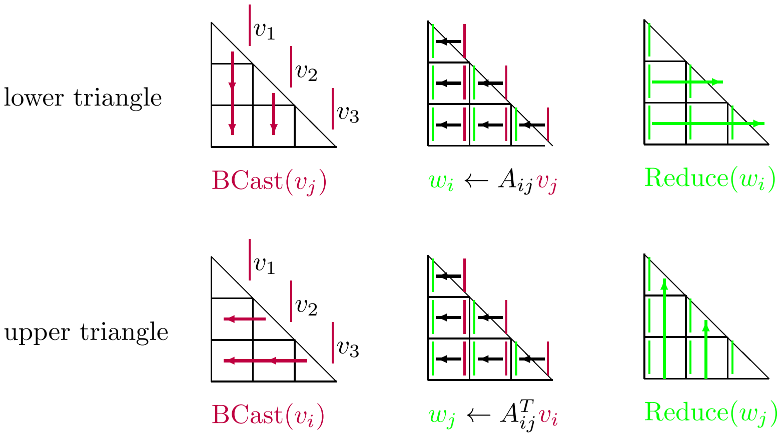
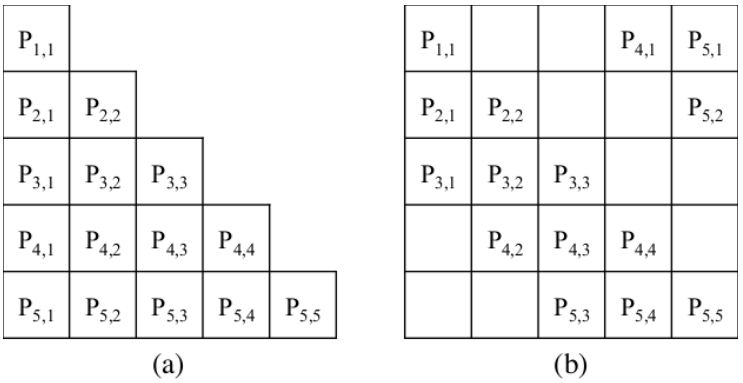


Figure 4: The decomposition of a symmetric matrix over a 2D triangular processing grid and the communication operations required during the parallel SpMV phase. Little triangular blocks along the matrix diagonal correspond to diagonal processors, and little squares correspond to off-diagonal processors. Each processor is responsible for communications and computations associated with the submatrix assigned to it.

The simple parallel SpMV scheme described above has a serious pitfall. Since different communication groups contain different number of processing units, the communication volume is not balanced among different communication groups. For example, the first column group contains n_d processing units, whereas the n_dth column group consists of a single processing unit only. This imbalance may cause significant load imbalances in large-scale computations. To address this, we extend the triangular grid on the left in Fig. 5 to a square grid on the right and place $n_d(n_d+1)/2$ processing units on this grid in such a way that each row or column of the square grid contains exactly $(n_d+1)/2$ processing units. (This is why we require n_d to be an odd integer.) We require that the processing unit that receives the $A_{i,j}$ submatrix to be placed on either the (i,j)th or the (j,i)th grid point, but not both. In particular, if the condition $i - j \leq (n_d+1)/2$ is satisfied, then the processing unit $P_{i,j}$ is placed on the (i,j)th grid point, otherwise it is placed onto the (j,i)th grid point. As a result, we can then create row and column communication groups with equal number of processing units based on the location of each processing unit on the square grid.

Although the above strategy ensures that the communication volume among different communication groups is balanced, the actual performance of the program will



$P_{1,1}$				
$P_{2,1}$	$P_{2,2}$			
$P_{3,1}$	$P_{3,2}$	$P_{3,3}$		
$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	
$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$

(a)

$P_{1,1}$			$P_{4,1}$	$P_{5,1}$
$P_{2,1}$	$P_{2,2}$			$P_{5,2}$
$P_{3,1}$	$P_{3,2}$	$P_{3,3}$		
	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	
		$P_{5,3}$	$P_{5,4}$	$P_{5,5}$

(b)

Figure 5: The layout of fifteen processing units (a) on a 2D lower triangular grid topology and (b) on a 5 × 5 square grid to balance row and column communication groups.

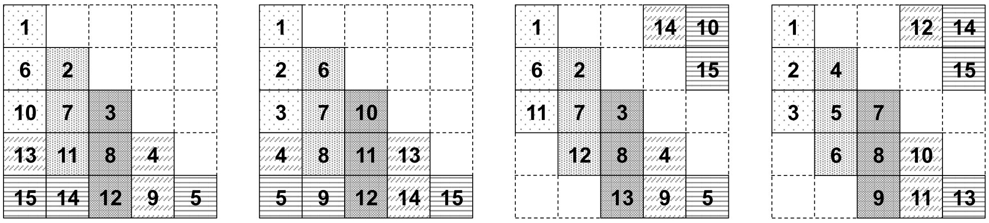


Figure 6: A schematic illustration of several different mapping schemes for a 5 × 5 grid, from left to right: DM, CM, BDM and BCM. Tasks mapped to the same column (row) of the grid belong to the same column (row) communication groups. Tasks with the same fill patterns belong to the same groups created for basis orthogonalization.

depend on other factors. The mapping between computational tasks and physical processing units has a strong influence, too.

There are many ways to map task blocks defined in Fig. 5 to different processors, see Fig. 6. The performance of different mapping schemes can be predicted from several metrics associated with a network load model that depends on the topology of the processor layout [11,12]. The task-to-process mapping should be defined in a way to minimize the effective load on the network measured in addition to communication volume imbalance. In particular, if we assign $1, 2, \ldots, n_{\mathrm{d}}$ to the diagonal processors first, and continue the assignment for each of the subdiagonals until all processors on the triangular grid are labeled, a scheme which we refer to as the diagonal-major (DM) ordering of processors, the measured performance of the parallel SpMV is relatively poor. On the other hand, if we go through the triangular processor grid column by column, and assign $1, 2, \ldots, n_{\mathrm{d}}$ along the way, which gives the column-major (CM) ordering, the performance of SpMV is much better.

6.2 Basis orthogonalization

To eliminate spurious eigenvalues [6], MFDn performs full orthogonalization among the columns of V in (7). As the number of columns in V increases, orthogonalization can become a computational bottleneck if it is not effectively parallelized.

For basis orthogonalization, we reconfigure the 2D triangular grid used for parallel SpMV into a $n_d \times (n_d + 1)/2$ rectangular processing grid as shown in Fig. 7(a). In an earlier version of MFDn [5], we used a 2D cyclic distribution of the columns of V, Fig. 7(b). In this scheme, communicating the local pieces of w, which we denote by w_i, among row communication groups of the 2D rectangular grid require expensive broadcast and reduction operations. We estimate the total communication volume to be $\mathcal{O}(n_d \times n)$ in this case. When the dimension of A is extremely large and the number of processing units used in the computation is large, this communication overhead significantly hinders the scalability of MFDn.

It turns out that we can reduce the communication overhead associated with basis orthogonalization by using a hierarchical 1D distribution of the basis vectors among all processing units. Note that each basis vector v was already divided into n_d subvectors v_j, $j = 1, 2, \ldots, n_d$ for SpMV computations, where each v_j is associated with the jth column group in the 2D square grid (Fig. 5). Each subvector v_j can then be partitioned further into $(n_d + 1)/2$ subvectors and distributed among processing units within the same column communication group, as shown in Fig. 7. In this case, the expensive broadcast operation required with the 2D cyclic distribution is replaced with a gathering operation (by `MPI_Gatherv`), which involves a communication volume of $\mathcal{O}(n)$. Similarly, the reduce operation after orthogonalization is replaced with a scattering operation (by `MPI_Scatterv`). The scattering operation involves $\mathcal{O}(n)$ data transfer, as well. As a result, when the basis vectors are partitioned hierarchically

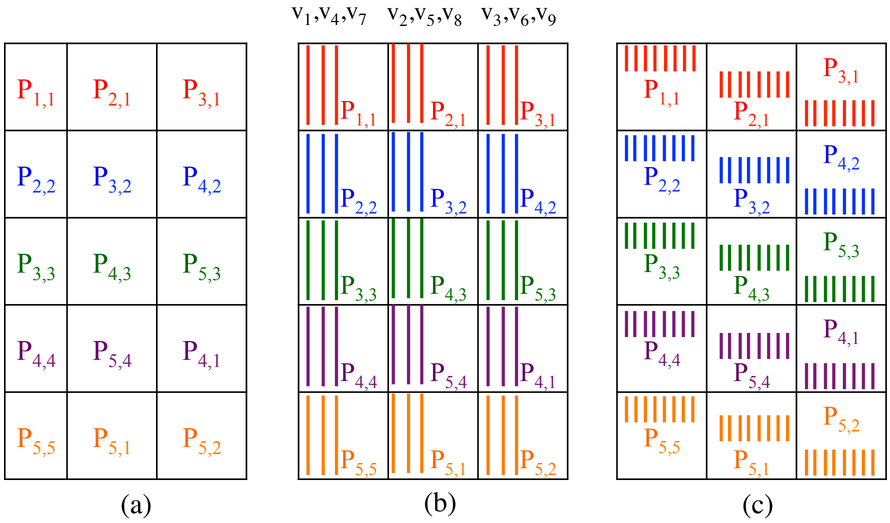


Figure 7: (a) 2D rectangular grid for 15 processors. (b) 2D cyclic distribution of the first 9 basis vectors in V over the 5×3 rectangular grid. (c) Hierarchical 1D distribution of the basis vectors in V. With this distribution, the vector w is first partitioned into 5 subvectors $\{w_i\}$, $i = 1, 2, \ldots, 5$, conformally with the column partitioning of A. Each subvector w_i is further partitioned into 3 shorter vectors, to be scattered to the processing units with matching labels.

in 1D, the total communication volume is $\mathcal{O}(n)$ instead of the $\mathcal{O}(n_d \times n)$ volume associated with the 2D cyclic distribution discussed earlier.

6.3 A hybrid MPI/OpenMP implementation

In recent years, distributed memory multi-core machines have become widely available for high performance computing. This trend is likely to continue in the foreseeable future. On such type of machine, a number of compute nodes are connected via a high speed communication network. Within each node, several processing units (or cores) share a common pool of memory. Such architecture allows us to reduce communication overhead and improve the throughput of computation by combining message passing based parallelization with thread based concurrency.

An effective technique for reducing communication volume is to restrict MPI communication among nodes that do not share a common pool of memory. This can be achieved through a hybrid programming paradigm, where local computations are performed in parallel using a thread based programming model such as OpenMP, and communication among nodes is done through MPI primitives. To perform a multi-threaded SpMV on a single node, we use the well-known compressed sparse column (CSC) method. The OpenMP parallelization for SpMV computations is performed at the outer-loop level. The columns of the sparse submatrix are assigned to OpenMP threads dynamically in chunks. The chunk size is chosen large enough to minimize the dynamic load distribution overheads, while maintaining a good load balance among threads.

The main benefit of a hybrid parallel implementation is the reduction in the memory footprint of large-scale computations [13]. Another important benefit is the reduced communication volume during the broadcast of v and reduction of w vectors, which is $\mathcal{O}(n \times n_d)$. This implies that for the same computation (so n is fixed), reducing the number of diagonal processing units n_d would lead to reduced communication overheads. Since $n_d \approx \sqrt{2n_p}$, by defining a processing unit to be a single CPU with t cores while fixing the total number of cores used at n_p, we effectively reduce the number of MPI processes by a factor of $\sqrt{t}$. Consequently, using hybrid MPI/OpenMP parallelization, the overall communication volume is reduced by a factor of $\sqrt{t}$ compared to a pure MPI implementation.

In the case of basis orthogonalization, the dense matrix vector multiplication performed on each node can be performed by simply calling a thread-enabled BLAS subroutine `dgemv`, which is now standard in the math libraries of multi-core platforms. Since the amount of data transfer required in a hierarchical 1D distributed basis orthogonalization scheme is $\mathcal{O}(n)$ and independent of the number of processing units, there is no reduction in the communication volume in this phase.

In addition to reducing communication volume, a hybrid OpenMP/MPI implementation on a distributed multi-core system also provides opportunities for hiding communication overhead by overlapping communication with computation. It is possible to implement a symmetric SpMV with a single pass over the elements of the sparse matrix where both the lower triangular and upper triangular calculations are performed together. The key observation that allows us to hide communication during the SpMV phase is that the symmetric SpMV computations can be divided into two subtasks: $w_i = A_{ij}v_j$ (which corresponds to the lower triangular part in Fig. 4) and $w_j = A_{ij}^T v_i$ (the upper triangular part). Such a separation breaks certain data dependencies during SpMV computations. Now the input for the second subtask, v_i, is not required by the first subtask and the output of the first subtask, w_i, can be reduced independently of the second subtask. We should note that going over the matrix elements twice by dividing the SpMV into two subtasks will induce a performance penalty. But since the matrix elements are streamed sequentially from memory, the additional burden on the memory subsystem is low compared to the irregular accesses to vector elements during SpMV.

In a pure MPI implementation of the symmetric SpMV, each processing unit is responsible for both communication and computation. As shown in Fig. 8, in the absence of non-blocking collective MPI primitives, effective overlapping of communication and computation in a pure MPI implementation is not trivial.

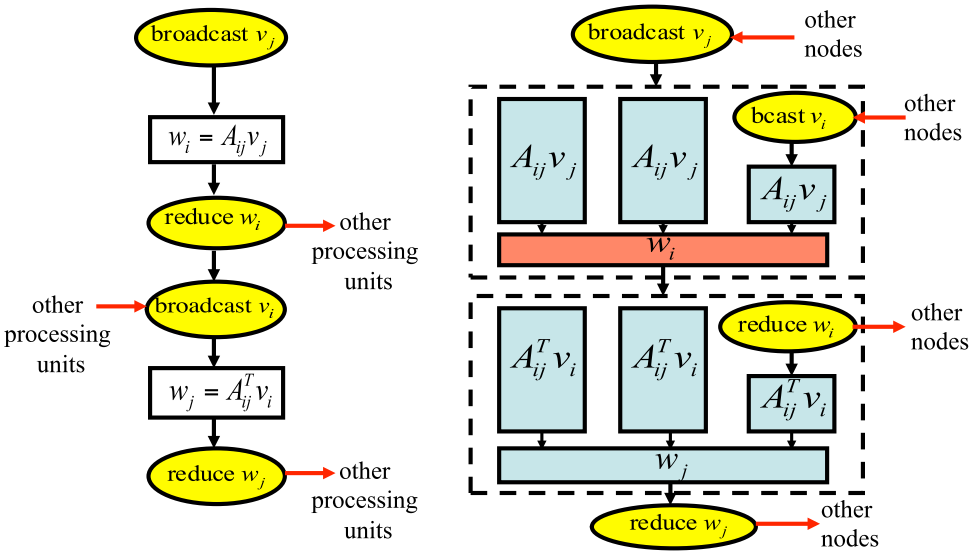


Figure 8: In a pure MPI implementation of a parallel symmetric SpMV on an off-diagonal processing unit (left subfigure), the communication blocks (yellow) separates the computational blocks (light blue). No communication/computation overlapping is possible. However, in a hybrid OpenMP-MPI implementation (right subfigure), the broadcast of v_i can be overlapped with the $w_i = A_{ij}v_j$ computation, and the reduction of w_j can be overlapped with the $w_j = A_{ij}^T v_i$ computation. The red blocks indicates where thread synchronization, which has very little overhead, is required.

However, in the hybrid OpenMP/MPI programming model where a processing unit runs on multiple cores, we may delegate one core (and the thread mapped to this core) to perform the communication, while other cores (and their corresponding threads) perform computations in parallel. As a result, the broadcast of v_i can be overlapped with the computations of the first subtask. Similarly, the reduction on w_i can be done while the computations of the second subtask are being still being carried out.

The hybrid OpenMP/MPI implementation of SpMV illustrated on the right in Fig. 8 hides the collective communications performed within row groups. When communication groups are created by using a column major ordering as discussed in section 6.1, communications along row groups tend to be the costliest ones because processing units that belong to the same row communication group are likely to be far apart from each other, as opposed to consecutively rank processing units within a column communication group [14]. Hence hiding communications within row groups is expected to have a large impact on the overall performance of the computation.

Note that in our scheme, there is no rigid designation of threads such as a communication thread or a computation thread. Since we dynamically schedule the computations among threads during SpMV, once the thread responsible for broadcasting v_i completes this communication task, it can join other threads in the multi-threaded computations of $w_i = A_{ij}v_j$. In addition, the reduction of w_i can be efficiently overlapped with the calculations of $w_j = A_{ij}^T v_i$ using the same technique, as shown in Fig. 8.

7 Computational example

In this section, we report performance gains achieved by incorporating the techniques discussed above in MFDn. We use one particular example to demonstrate performance gain. More performance results can be found in [11,12]. The test problem we selected is the ^{10}B nucleus. We are interested in computing 10 algebraically smallest eigenvalues of the Hamiltonian matrix $\hat{H}$ constructed by setting $N_{\max} = 8$ and $M_j = 1$. A 2-body interaction potential is used. The dimension of the matrix is roughly 4.8×10^8 and it contains roughly 1.2×10^{12} nonzero matrix elements in the lower triangular part of the matrix.

Table 1 lists five different implementations of a parallel Lanczos algorithm. They correspond to progressive improvements we made in terms of task to processor mapping, the way the Lanczos basis vectors are distributed, and whether there is any overlap between computation and communication. The pure MPI implementation where processes are arranged on a triangular grid in a diagonal major (DM) order

Table 1: Five versions of parallel implementations of the Lanczos algorithm. They differ by task to processing mapping, which is related to process ordering, the way the Lanczos basis vectors are distributed and whether there is any overlap between computation and communication.

Version	Parallelization Strategy	Process Ordering	Ortho. Scheme	Comm Overlapping
*ver*1	MPI only	DM	2D Cyclic	none
*ver*2	MPI/OpenMP	DM	2D Cyclic	none
*ver*3	MPI only	BCM	1D Hierarchical	none
*ver*4	MPI/OpenMP	BCM	1D Hierarchical	none
*ver*5	MPI/OpenMP	BCM	1D Hierarchical	row only

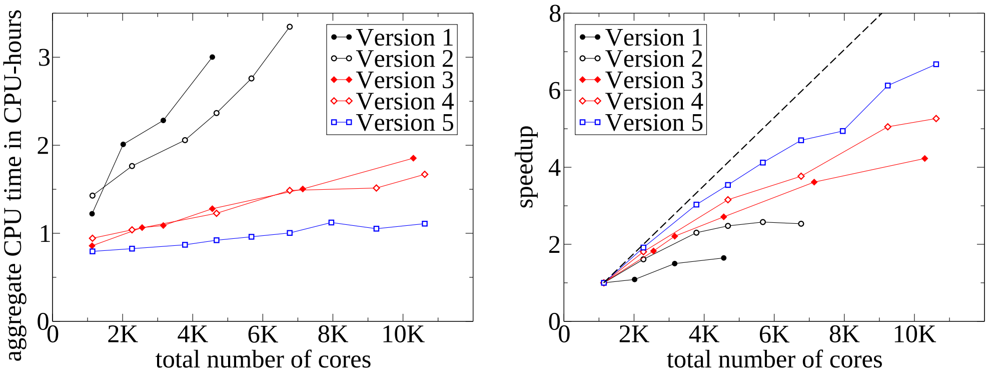


Figure 9: Strong scaling (left subfigure) and speed-ups (right subfigure) achieved by different versions on the ^{10}B, $N_{\max}$=8, M_j=1 testcase starting from 1,140 cores on up to 10,560 cores.

and the Lanczos vectors are distributed on a 2D rectangular grid in a cyclic fashion [5] is defined as $ver1$. On the other extreme, $ver5$ contains all techniques we discussed above. In particular, it allows the SpMV computations to be overlapped with communication.

The left subfigure in Fig. 9 shows the scalability of each version in Table 1 for this test problem. We define the CPU-hour cost of a single Lanczos iteration as the wall-clock time (in hours) required for an iteration times the number of cores used in that run. So in Fig. 9, the CPU-hour cost plot of an implementation with perfect strong-scaling properties would follow a horizontal line. As seen in Fig. 9, $ver5$ follow a nearly horizontal line, meaning that it has very good strong-scaling properties.

The right subfigure in Fig. 9 shows the speed-up achieved by each version for the same testcase. The speed-up measurements from $ver1$ and $ver2$ plateau after about 4,000 cores. Similarly, the gains from $ver3$ and $ver4$ start increasing only slightly after 7,000 and 9,000 cores, respectively. However, we are still able to achieve significant speed-up beyond 10,000 cores in $ver5$.

Acknowledgements

This work was supported in part by the National Science Foundation under Grant No. PHY-0904782 and the Department of Energy under Grant Nos. DE-FG02-87ER40371 and DESC0008485 (SciDAC-3/NUCLEI). The computational resources were provided by the National Energy Research Scientific Computing Center (NERSC), which is supported by the DOE Office of Science under Contract No. DE-AC02-05CH11231.

References

[1] J.P. Vary, *The many-fermion dynamics shell-model code.* 1992 (*unpublished*).

[2] J.P. Vary and D.C. Zheng, *The many-fermion dynamics shell-model code.* 1994 (*unpublished*).

[3] Z. Gan, Y. Alexeev, M. S. Gordon and R. A. Kendall, J. Chem. Phys. **119**, 47–59 (2003).

[4] E. Rossi, G. L. Bendazzoli and S. Evangelisti, J. Comp. Chem. **19(6)**, 658–672 (1998).

[5] P. Sternberg, E. G. Ng, C. Yang, P. Maris, J. P. Vary, M. Sosonkina and H. V. Le, *Accelerating configuration interaction calculations for nuclear structure*, in *Proc. 2008 ACM/IEEE Conf. on Supercomputing.* IEEE Press, Piscataway, NJ, 2008, p. 15:1.

[6] B. N. Parlett, *The Symmetric Eigenvalue Problem.* Prentice-Hall, 1980.

[7] D. C. Sorensen, SIAM J. Matrix Anal. Appl. **13(1)**, 357–385 (1992).

[8] R. B. Lehoucq, D. C. Sorensen and C. Yang, *ARPACK users guide: Solution of large scale eigenvalue problems by implicitly restarted Arnoldi methods.* SIAM, Philadelphia, PA, 1998.

[9] E. R. Davidson, J. Comput. Phys. **17**, 87–94 (1975).

[10] A. Knyazev, SIAM J. Sci. Comput. **22(2)**, 517–541 (2001).

[11] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris and J. P. Vary, *Topology-aware mappings for large-scale eigenvalue problems*, in *Euro-Par*, eds. C. Kaklamanis, T. S. Papatheodorou and P. G. Spirakis. Lecture Notes Comput. Sci. **7484**, 830 (2012).

[12] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris and J. P. Vary, *Improving the scalability of symmetric iterative Eigensolver for multi-core platforms*, Concurrency Computat.: Pract. Exper. (2013), DOI: 10.1002/cpe.3129.

[13] P. Maris, M. Sosonkina, J. P. Vary, E. G. Ng and C. Yang, Proc. Comput. Sci. **1**, 97–106 (2010).

[14] C. Albing, N. Troullier, S. Whalen, R. Olson, J. Glenski, H. Pritchard and H. Mills, *Scalable node allocation for improved performance in regular and anisotropic 3d torus supercomputers*, in *Recent Advances in the Message Passing Interface.* LNCS **6960**, 61–70 (2011).