7-2003

The Parallel Implementation of a Full Configuration Interaction Program

Zhengting Gan
Iowa State University

Yuri Alexeev
Iowa State University

Mark S. Gordon
Iowa State University, mgordon@iastate.edu

Ricky A. Kendall
Iowa State University

The Parallel Implementation of a Full Configuration Interaction Program

Abstract
Both the replicated and distributed data parallel full configuration interaction (FCI) implementations are described. The implementation of the FCI algorithm is organized in a hybrid strings-integral driven approach. Redundant communication is avoided, and the network performance is further optimized by an improved distributed data interface library. Examples show linear scalability of the distributed data code on both PC and workstation clusters. The new parallel implementation greatly extends the hardware on which parallel FCI calculations can be performed. The timing data on the workstation cluster show great potential for using the new parallel FCI algorithm in expanding applications of complete active space self-consistent field applications.

Keywords
Configuration interaction

Disciplines
Chemistry

Comments
The following article appeared in *Journal of Chemical Physics* 119 (2003): 47, and may be found at doi:10.1063/1.1575193.

The parallel implementation of a full configuration interaction program

Zhengting Gan, Yuri Alexeev, Mark S. Gordon, and Ricky A. Kendall

Citation: The Journal of Chemical Physics **119**, 47 (2003); doi: 10.1063/1.1575193
View online: http://dx.doi.org/10.1063/1.1575193
View Table of Contents: http://scitation.aip.org/content/aip/journal/jcp/119/1?ver=pdfcov
Published by the AIP Publishing

JOURNAL OF CHEMICAL PHYSICS VOLUME 119, NUMBER 1 1 JULY 2003

The parallel implementation of a full configuration interaction program

Zhengting Gan
Scalable Computing Laboratory, Ames Laboratory U.S. D.O.E., Ames, Iowa 50011

Yuri Alexeev and Mark S. Gordon
Scalable Computing Laboratory, Ames Laboratory U.S. D.O.E., Ames, Iowa 50011 and Department of Chemistry, Iowa State University, Ames, Iowa 50011

Ricky A. Kendall
Scalable Computing Laboratory, Ames Laboratory U.S. D.O.E., Ames, Iowa 50011 and Department of Computer Science, Iowa State University, Ames, Iowa 50011

(Received 26 December 2002; accepted 26 March 2003)

Both the replicated and distributed data parallel full configuration interaction (FCI) implementations are described. The implementation of the FCI algorithm is organized in a hybrid strings-integral driven approach. Redundant communication is avoided, and the network performance is further optimized by an improved distributed data interface library. Examples show linear scalability of the distributed data code on both PC and workstation clusters. The new parallel implementation greatly extends the hardware on which parallel FCI calculations can be performed. The timing data on the workstation cluster show great potential for using the new parallel FCI algorithm in expanding applications of complete active space self-consistent field applications. © *2003 American Institute of Physics.* [DOI: 10.1063/1.1575193]

I. INTRODUCTION

Full configuration interaction (FCI) provides the exact nonrelativistic solution of the many-electron Schrödinger equation in a given finite one-electron basis space. Therefore it is commonly used as a benchmark tool to assess approximate correlation methods. In addition, FCI is also employed in complete-active space self-consistent field (CASSCF) (Ref. 1) programs to obtain the CI coefficients of CASSCF wave functions. Because the FCI calculation is carried out at every CASSCF iteration, considerable effort has been expended to develop and implement efficient FCI algorithms to handle the relatively large active spaces, required by complex chemical systems.[1–5]

Because of the simplicity in computing the coupling coefficients, almost all modern FCI algorithms are based on determinants instead of configuration state functions (CSFs). A major step forward in determinant-based FCI was introduced by Handy[2] with the idea of separate alpha and beta strings for alpha and beta spins. Following this innovation, more efficient algorithms have been proposed and implemented, including the work by Olsen *et al.*,[3] Zarrabian *et al.*,[4] Michael *et al.*,[5] Rossi *et al.*,[6] Sherrill *et al.*,[7] Ivanic *et al.*,[9] and others. However, despite the diversity of the algorithms, the implementations of FCI programs remain essentially in two different ways, either the integral driven approach or the configuration driven approach.[6,8]

The memory and efficiency requirements of a FCI calculation make it an ideal application for parallel computing. By taking advantage of the computational capacity provided by state-of-the-art high performance parallel supercomputers, FCI calculations that include up to ten billion determinants have been reported.[8] However, the problem of parallel full CI performance still requires attention. For example, due to the excessive communication present in parallel FCI calculations, existing parallel FCI implementations are limited to supercomputers, such as the CRAY-T3E[8,17] and IBM SP.[6] Because clusters of PCs and workstations have become increasingly accepted as powerful alternatives to expensive supercomputers, a FCI code that runs effectively on clusters is highly desired, albeit a challenging, task.

The difficulties in achieving high performance on parallel computers come from the complexity of modern computer architectures, and the networked distributed memory makes the situation even worse. Since there are multiple performance factors on parallel computers, any single mode CI driven method will have some pitfalls. Instead, a more flexible algorithm is needed. The original determinant FCI code[9] in GAMESS is a straight string driven implementation. The algorithm produces reasonable efficiency, but the computational potential is not fully exploited, and the structure of data and computational algorithm is not well suited for distributed computing. In this paper we present an efficient implementation of both replicated data and distributed data parallel FCI algorithms. In these new codes a combined string driven and integral driven approach is proposed to optimize both the sequential performance and the interprocessor communication. In the distributed data implementation we identify the redundant communication in a parallel FCI calculation and its collective nature. These problems are solved conveniently in our combined approach. In addition, the communication performance is further improved by the implementation of new distributed data interface (DDI)[10] communication routines. The performance and scalability of the new parallel implementation are demonstrated by examples.

II. THEORY AND METHODS

A. Description of the FCI algorithm

The FCI solves the Schrödinger equation by making use of the variational principle. In practice, it solves the eigenvector problem of the Hamiltonian matrix in the determinant basis space defined by distributing the n electrons over N spin orbitals. Since usually only a few eigenvalues and eigenvectors are required in CI calculations, the eigenvalue problem is commonly solved by iterative diagonalization methods proposed by Davidson[11] or Liu.[12] In the Davidson diagonalization method the Hamiltonian matrix is projected onto a small set of subspace vectors, and the solution vector is obtained by full diagonalization in this small subspace. The subspace is improved at each iteration by adding a new basis vector, which is derived from the Ritz residual. When the subspace becomes too large to handle, a restart procedure is applied and the subspace is contracted. The most time consuming step in each iteration is the calculation of the Hamiltonian projection vector σ for the new basis vector C,

$$\sigma=HC, \tag{1}$$

where H is the Hamiltonian matrix. As the dimension of the CI space increases, the storage of the Hamiltonian matrix in memory or even on disk soon becomes impossible. Thus a direct CI technique must be employed to calculate the σ vector without explicit construction of the Hamiltonian matrix. This strategy eliminates the storage problem of the Hamiltonian matrix, so a significantly larger CI calculation becomes possible.

The Hamiltonian operator $\hat{H}$ in the Schrödinger equation is generally expressed in second quantized form as

$$\hat{H}=\sum_{ij} h_{ij}\hat{E}_{ij}+\frac{1}{2}\sum_{ijkl}(ij,kl)(\hat{E}_{ij}\hat{E}_{kl}-\delta_{jk}\hat{E}_{il}). \tag{2}$$

Here $\hat{E}_{ij}$ is the shift operator, h_{ij} are one-electron Hamiltonian matrix elements, and (ij,kl) are two-electron integrals. In the following discussions we will use u,v,w to represent determinants, I,J,K to represent strings, k,l to represent the orbital array of a string, L_α,L_β to represent alpha and beta excitation lists, and superscripts λ,τ to denote unspecified alpha and beta spins.

In the FCI wave function, a determinant is expressed as a combination of alpha and beta strings. An alpha string is defined as an ordered product of creation operators for spin orbitals with alpha spin, and a beta string is defined similarly,

$$|K\rangle=|K_\alpha K_\beta\rangle,\quad |K_\alpha\rangle=\prod_{i=1}^{n_\alpha}\alpha^{+}_{l(i),\alpha},\quad |K_\beta\rangle=\prod_{i=1}^{n_\beta}\alpha^{+}_{k(i),\beta}. \tag{3}$$

$l(i)$ and $k(i)$ are the orbital arrays for the alpha and beta strings, respectively. It is convenient to split $\hat{E}_{ij}$ into alpha and beta parts,

$$\hat{E}_{ij}=\hat{E}^{\alpha}_{ij}+\hat{E}^{\beta}_{ij}, \tag{4}$$

and the Hamiltonian can be rewritten as a combination of six parts,

$$\begin{aligned}
&\hat{H}=\hat{H}_1+\hat{H}_2+\hat{H}_3+\hat{H}_4+\hat{H}_5+\hat{H}_6,\\
&\hat{H}_1=\sum_i h_{ii}\hat{E}^{\alpha}_{ii}+\frac{1}{2}\sum_{ij}[(ii,jj)-(ij,ji)]\hat{E}^{\alpha}_{ii}\hat{E}^{\alpha}_{jj}\\
&\quad+\frac{1}{2}\sum_{ij}(ii,jj)\hat{E}^{\alpha}_{ii}\hat{E}^{\beta}_{jj}+\sum_i h_{ii}\hat{E}^{\beta}_{ii}\\
&\quad+\frac{1}{2}\sum_{ij}[(ii,jj)-(ij,ji)]\hat{E}^{\beta}_{ii}\hat{E}^{\beta}_{jj}\\
&\quad+\frac{1}{2}\sum_{ij}(ii,jj)\hat{E}^{\beta}_{ii}\hat{E}^{\alpha}_{jj},\\
&\hat{H}_2=\sum_{i\neq j} h_{ij}\hat{E}^{\alpha}_{ij}+\frac{1}{2}\sum_{iijk,i\neq j}[(ij,kk)-(ik,kj)]\hat{E}^{\alpha}_{ij}\hat{E}^{\alpha}_{kk}\\
&\quad+\frac{1}{2}\sum_{ijk,i\neq j}(ij,kk)\hat{E}^{\alpha}_{ij}\hat{E}^{\beta}_{kk},\\
&\hat{H}_3=\sum_{i\neq j} h_{ij}\hat{E}^{\beta}_{ij}+\frac{1}{2}\sum_{ijk,i\neq j}[(ij,kk)-(ik,kj)]\hat{E}^{\beta}_{ij}\hat{E}^{\beta}_{kk}\\
&\quad+\frac{1}{2}\sum_{ijk,i\neq j}(ij,kk)E^{\beta}_{ij}\hat{E}^{\alpha}_{kk},\\
&\hat{H}_4=\frac{1}{2}\sum_{ijkl,i\neq j\neq k\neq l}(ij,kl)(\hat{E}^{\alpha}_{ij}\hat{E}^{\alpha}_{kl}),\\
&\hat{H}_5=\frac{1}{2}\sum_{ijkl,i\neq j\neq k\neq l}(ij,kl)(\hat{E}^{\beta}_{ij}\hat{E}^{\beta}_{kl}),\\
&\hat{H}_6=\frac{1}{2}\sum_{ijkl,i\neq j\neq k\neq l}(ij,kl)(\hat{E}^{\alpha}_{ij}\hat{E}^{\beta}_{kl}+\hat{E}^{\beta}_{ij}\hat{E}^{\alpha}_{kl}).
\end{aligned} \tag{5}$$

Here some of the two electron terms are grouped with one electron term for ease of implementation. The six terms can be viewed as the diagonal term ($\hat{H}_1$), alpha ($\hat{H}_2$) and beta ($\hat{H}_3$) single replacement terms, alpha ($\hat{H}_4$) and beta ($\hat{H}_5$) double replacement terms, and the combination of alpha and beta single excitations ($\hat{H}_6$).

The coupling coefficients are the matrix elements of operator $\hat{E}^{\lambda}_{ij}$ and the operator product $\hat{E}^{\lambda}_{ij}\hat{E}^{\tau}_{kl}$.

$$\begin{aligned}
&E^{\lambda,uv}_{ij}=\langle u|\hat{E}^{\lambda}_{ij}|v\rangle,\\
&E^{\lambda\tau,uv}_{ij,kl}=\langle u|\hat{E}^{\lambda}_{ij}\hat{E}^{\tau}_{kl}|v\rangle.
\end{aligned} \tag{6}$$

In the determinant basis, a nonzero coupling coefficient of a single operator $\hat{E}^{\lambda}_{ij}$ can be either 1 or -1, and the associated phase factor $P^{\lambda,uv}_{ij}$ can be obtained by counting the occupied orbitals that u and v have in common between orbitals i and j, as discussed in Ref. 8.

$$\begin{aligned}
&E^{\lambda,uv}_{ij}=(-1)^{P^{\lambda,uv}_{ij}},\\
&P^{\lambda,uv}_{ij}=\sum_{k=i+1}^{j-1}\hat{E}^{\lambda,uv}_{kk}.
\end{aligned} \tag{7}$$

The coefficients $E^{\lambda\tau,uv}_{ij,kl}$ can be evaluated by summing over all the intermediate states,

$$E^{\lambda\tau,u\nu}_{ij,kl}=\sum_{w}\langle u|\hat{E}^{\lambda}_{ij}|w\rangle\langle w|\hat{E}^{\tau}_{kl}|\nu\rangle. \tag{8}$$

Although this may appear to be very complicated, only one intermediate state of w has a nonzero contribution, so it is computed as the product of two single operator coefficients, and its phase factor is the sum of the two,

$$E^{\lambda\tau,u\nu}_{ij,kl}=(-1)^{P^{\lambda\tau,u\nu}_{ij,kl}};\quad P^{\lambda\tau,u\nu}_{ij,kl}=P^{\lambda,uw}_{ij}+P^{\tau,w\nu}_{kl}. \tag{9}$$

Now, let H_i be the matrix form of operator $\hat{H}_i$, with the computational task of the σ vector apportioned accordingly,

$$\begin{aligned}
\sigma&=H_1C+H_2C+H_3C+H_4C+H_5C+H_6C\\
&=\sigma_1+\sigma_2+\sigma_3+\sigma_4+\sigma_5+\sigma_6,\\
\sigma_{1,u}&=\sum_{\nu}\Bigg\{\sum_{i}h_{ii}E^{\alpha,u\nu}_{ii}+\sum_{i<j}[(ii,jj)-(ij,ji)]E^{\alpha,u\nu}_{ii}E^{\alpha,u\nu}_{jj}\\
&\quad+\sum_{ij}(ii,jj)E^{\alpha,u\nu}_{ii}E^{\beta,u\nu}_{jj}+\sum_{i}h_{ii}E^{\beta,u\nu}_{ii}\\
&\quad+\sum_{i<j}[(ii,jj)-(ij,ji)]E^{\beta,\mu\nu}_{ii}E^{\beta,u\nu}_{jj}\Bigg\}C_{\nu},\\
\sigma_{2,u}&=\sum_{\nu}\Bigg\{\sum_{i\neq j}h_{ij}E^{\alpha,u\nu}_{ij}+\sum_{ijk,i<j}[(ij,kk)-(ik,kj)]\\
&\quad\times E^{\alpha,u\nu}_{ij}E^{\alpha,u\nu}_{kk}+\sum_{ijk,i<j}(ij,kk)E^{\alpha,u\nu}_{ij}E^{\beta,u\nu}_{kk}\Bigg\}C_{\nu},\\
\sigma_{3,u}&=\sum_{\nu}\Bigg\{\sum_{i\neq j}h_{ij}E^{\beta,u\nu}_{ij}+\sum_{ijk,i<j}[(ij,kk)-(ik,kj)]\\
&\quad\times E^{\beta,u\nu}_{ij}E^{\beta,u\nu}_{kk}+\sum_{ijk,i<j}(ij,kk)E^{\beta,u\nu}_{ij}E^{\alpha,u\nu}_{kk}\Bigg\}C_{\nu}, \qquad (10)\\
\sigma_{4,u}&=\sum_{\nu}\Bigg\{\sum_{i\neq j\neq k\neq l,ij<kl}[(ij,kl)-(il,jk)]E^{\alpha\alpha,u\nu}_{ij,kl}\Bigg\}C_{\nu},\\
\sigma_{5,u}&=\sum_{\nu}\Bigg\{\sum_{i\neq j\neq k\neq l,ij<kl}[(ij,kl)-(il,jk)]E^{\beta\beta,u\nu}_{ij,kl}\Bigg\}C_{\nu},\\
\sigma_{6,u}&=\sum_{\nu}\Bigg\{\sum_{i\neq j\neq k\neq l}(ij,kl)(E^{\alpha\beta,u\nu}_{ij,kl})\Bigg\}C_{\nu},
\end{aligned}$$

where u and ν denote elements of vectors σ and C. Following Olsen,[3] in the case of $Ms=0$ the computation of terms σ_3 and σ_5 can be eliminated and the computational effort of σ_6 can also be halved by imposing the condition

$$\sigma(\alpha,\beta)=(-1)^{s}\sigma(\beta,\alpha). \tag{11}$$

B. String and list generation

A string is represented by a list of occupied orbitals in ascending energy order. During a FCI calculation, the information about a string is frequently accessed, so it is expedient to store this information rather than generate it on the fly. An effective way to accomplish this is to store strings in their bit format. In a system containing n_α α electrons and M orbitals, a string can be represented by a binary word of length M. Each bit represents an orbital; the bit is set to 1 if the orbital is occupied and set to 0 if it is empty. This is also called a string's binary representation. In FORTRAN a string can be stored as an integer and its information can be easily retrieved using the *ibtest* intrinsic function. However, if the number of orbitals M exceeds the total number of the bits of an integer word, this storage scheme becomes very inconvenient. First, a string has to be stored using several integers and the bit testing becomes nontrivial. Second, the efficiency of bit testing is also reduced because of the increased number of unoccupied orbitals. To avoid such problems an alternative orbital format is used in the code presented here, when the orbital space is relatively large. In this scheme only the occupied orbitals of a string are stored and each takes one byte of memory. In FORTRAN this can be implemented using an *INTEGER*1* or *CHARACTER* array. The combination of bit format and orbital format exploits the nature of FCI calculations, as a realistic FCI calculation can either have a large number of electrons or a large number of active orbitals, but not both. Thus either the bit format or the orbital format of a string will be suitable for a given FCI calculation.

The canonical order of a string K_λ can be obtained by using a modified *WYB* series arrays proposed by Wang and co-workers (*WYB*),[13,14]

$$\mathrm{Cano}(K_\lambda)=1+\sum_{i=1}^{n_\lambda}WYB_i[k(i)]. \tag{12}$$

Here λ denotes either alpha or beta spin of electrons, n_λ is the number of electrons in string K_λ, and $k(i)$ is the orbital array. The *WYB* arrays can be generated recursively,

$$WYB_i(J)=WYB_{i-1}(J-1)+WYB_i(J-1), \tag{13}$$

or be calculated directly

$$WYB_i(J)=\binom{J-1}{i}. \tag{14}$$

The canonical order generated by this indexing system is different from the addressing system proposed by Knowles *et al.*,[15] where the Z matrix is used. Here we use $L(n_\alpha,M)$ to denote the list of alpha strings in canonical order, n_α is the number of electrons with alpha spin and M is the total number of orbitals. Table I shows the list of $n_\alpha=3$, $M=6$ using the two different indexing schemes. Only one set of *WYB* arrays is used for both alpha and beta strings, while the Z matrix has to be built for alpha and beta strings separately. Besides, the Z matrix is more complicated to construct compared to *WYB* arrays. The formula for the Z matrix is

$$\begin{aligned}
Z(i,J)&=\sum_{m=M-J+i}^{M-i}\left[\binom{m}{n_\alpha}-\binom{m-1}{n_\alpha-i-1}\right],\\
&\quad(M-n_\alpha+i\geqslant J\geqslant i;i<n_\alpha),\\
Z(n_\alpha,J)&=J-n_\alpha,\quad(M\geqslant J\geqslant n_\alpha).
\end{aligned}$$

The canonical order of alpha and beta strings is used to address the determinants in the CI vector. The structure of the FCI coefficient matrix can be viewed as a two-dimensional matrix whose columns and rows are labeled by the canonical

TABLE I. The order of alpha strings using *WYB* arrays and the *Z* matrix for $M=6$, $n_\alpha=3$.

Order of string K_α	String K_α	
	WYB array	*Z* matrix
1	123	123
2	124	124
3	134	125
4	234	126
5	125	134
6	135	135
7	235	136
8	145	145
9	245	146
10	345	156
11	126	234
12	136	235
13	236	236
14	146	245
15	246	246
16	346	256
17	156	345
18	256	346
19	356	356
20	456	456

order of alpha and beta strings, respectively. When Abelian spatial symmetry is exploited, we define the irreducible representation (irrep) of a string as the product of the irreps of its occupied orbitals,

$$\Gamma(K_\lambda)=\prod^{i=1,n_\lambda}\Gamma[k(i)]. \tag{15}$$

Here $\Gamma(K)$ and $\Gamma(k)$ represent the irreps of string K and orbital k, respectively. Because of the spatial symmetry imposed on the CI coefficient vector, the allowed combination of alpha and beta strings must be symmetry matched. If alpha and beta strings are grouped according to their irreps, the matrix of CI vectors becomes symmetry blocked. Each block consists of only the alpha columns of that given irrep and thus has reduced dimension. Let C_α and C_β represent the canonical order of string K_α and K_β. Then, the address of a determinant is obtained by

$$Addr(K_\alpha,K_\beta)=ADDR(C_\alpha)+ORDB(C_\beta), \tag{16}$$

where the entry address of each alpha column is kept in the *ADDR* array, and the *ORDB* array maps the canonical order of the beta string onto the actual order in its symmetry block, i.e., the offset value in the alpha column it belongs to. The contribution from the alpha string and the beta string can be computed and stored separately; this facilitates the combination of alpha and beta strings.

The simple formula for calculating a string's canonical order and the addressing scheme for a CI vector also leads to the optimization of a list generation technique. Given a string K_λ and operator $\hat{E}_{ij}$, its single excitation string K'_λ can be generated by removing the electron from the jth orbital and placing it in the ith orbital. The canonical order of K'_λ and the phase factor of $\hat{E}_{ij}$ can be computed once the string K'_λ is formed. However, since only the canonical order C'_λ and the phase factor are of interest, it is possible to obtain them without explicit construction of the new string. This can be done by looping over the occupied orbitals of K_λ in strictly ascending order with new orbital i included and old orbital j excluded. The phase factor is computed at the same time by counting the number of occupied orbitals between orbital i and j. The entire operation is completed within one single loop over the original string, and no new string is formed. So, the operation count is basically cut by half. This optimized list generation technique is very useful because list generation is used extensively in FCI algorithms.

The list generation algorithm described here is designed to serve the needs of the combined CI driven approach, in which both string driven and integral driven loops will be employed. For this reason, a buffer array is used to hold the generated excitation list. The single excitation string driven list is straightforwardly generated by applying all possible $\hat{E}_{ij}$ operators to each string. To generate the double excitation list, operators $\hat{E}_{ij}$ and $\hat{E}_{kl}$ are applied separately to each string. The intermediate string K'_λ after applying $\hat{E}_{kl}$ is generated explicitly, so the final double excitation strings K''_λ can be computed from K'_λ, applying the technique discussed above. The method presented here does not generate the integral driven strings directly. If an integral driven list is desired, it is obtained by sorting the stored string driven list. It should be mentioned that various advanced integral driven list generation methods have been proposed, including the direct list approach[5,8] and reduced list approach.[6] The possibility of employing such algorithms has also been considered. The problem with the direct list approach is that two entire string space tests are required for each pair of integral indices (i,j), and most such test operations are redundant. In addition, the enormous bit counting operation in the direct list approach raises a technical concern because such an operation can be very inefficient in FORTRAN unless mixed *c*/Fortran programming is used. The problem of string space testing is avoided in the more recently proposed reduced list approach. In the latter, the excitation list is generated by inserting a pair of occupied orbital i and unoccupied orbital j into the reduced $n-1$ electron list. All the generated strings are valid excitation pairs, thus no operation is wasted. However, this algorithm cannot guarantee the symmetry of the generated strings, since the reduced list no longer has any symmetry information. Such problems do not exist in the list generation algorithm presented here, as the imposed symmetry requirement on strings can be easily satisfied by looping over only the symmetry matched orbitals. Let Γ denote the irrep, i and j denote the orbitals, and K'_λ the strings generated from K_λ by applying E_{ij}. We have

$$\Gamma(K_\lambda)^*\Gamma(K'_\lambda)=\Gamma(i)^*\Gamma(j). \tag{17}$$

The orbital pair i,j can simply be skipped if it does not match the required symmetry. Since the string driven list generation is highly efficient, it is more advantageous to obtain the integral driven excitation list from the string driven one at the expense of extra sorting and storage, rather than adopt the list algorithms discussed above. Another benefit of this method is that the spatial locality in the string driven list is partially preserved in the final integral driven list, thus reducing communication.

C. Algorithm implementation

The σ vector updating step is traditionally organized in either an integral-driven or a string-driven approach, as discussed in Refs. 6 and 8. However, the choice of one of these methods has mostly been concerned with the operation count, not the data structure and its access pattern in CI calculations. Because of the nonuniform memory access (NUMA) pattern of modern CPU architectures, one generally needs to exploit either spatial locality or temporary data locality in order to obtain reasonable CPU performance. In parallel computing, the distributed memory adds another layer to the memory hierarchy. This provides huge storage capacity, but generally incurs high latency and has limited bandwidth to access. Clearly, exploiting the data locality is at least as important as decreasing the operation count to achieve high performance on modern hardware architectures. With these considerations in mind, a combined string driven and integral driven approach has been implemented in the σ vector construction. The combined multiple CI driven scheme was first proposed in the GUGACI code developed by one of us,[16] where the main data structure consists of three parts: the CI coefficient vector, the integral array, and the coupling coefficients. Accordingly, the configuration driven, integral driven, and loop driven approaches are integrated at different loop levels to make memory access to each part more efficient. In a FCI calculation the primary data is contained in the C and σ vectors, so the string driven loops should be employed wherever these vectors are involved. In a sequential code, the string driven loops are best placed in the innermost loop updating the σ vector. In a distributed data parallel code the string driven loops can also be used in the outermost loop to optimize the communication between processors. Although the storage of the integral array is generally not a problem in a FCI calculation, the integral driven approach is much more efficient than the string driven one in computing Hamiltonian matrix blocks. In the integral driven approach the same intermediate result can be used to compute a series of similar Hamiltonian matrix blocks, which is actually generated implicitly in a direct CI approach. Reusing data here is a powerful technique, because it not only saves computation time, but also exploits the temporary data locality at the same time. It is thus desirable to integrate the two schemes together so that both the construction of the Hamiltonian matrix and the matrix vector product can be carried out with optimal performance. We now briefly describe the implementation of the combined CI driven scheme in the two major routines, the beta-beta routine and the alpha-beta routine.

The beta-beta routine computes the terms of σ_3 and σ_5 [Eq. (10)], which correspond to the single and double replacement of beta strings, respectively. In the algorithm presented here part of the two-electron operator term that contributes to the same determinant pairs is also grouped into σ_3. Such grouping is expected to be much more efficient than computing those terms separately. The contribution to the same Hamiltonian matrix element is added together before the multiplication by CI coefficients. As a result, the number of multiplication operations is reduced, and (more important) fewer memory *load* and *store* operations on the CI elements are required. In the straightforward string driven implementation of σ_3 one loops over each beta string and generates its single replacement strings first, then each beta pair is combined with the corresponding alpha strings and their contributions to σ_3 can be counted. However, this simple algorithm does not perform well. One reason is that the stride of the inner loop, which equals the number of beta strings in that symmetry block, is too large. In modern CPU architecture a large stride in the innermost loop results in poor cache performance. Most importantly, there is repeated computation. Since the detailed orbital information of each alpha string is needed to compute the two-electron term in σ_3 for each beta pair the entire alpha strings must be unpacked. Despite the optimized string storage scheme, such excessively repeated operations would still cost considerable computer time. Moreover, the computation of Hamiltonian elements is inefficient as well, because every element is computed from scratch.

To address the problems mentioned above, an integral driven approach is employed inside the string driven outer loop. First, one sets up a buffer array to store the generated beta single replacement pairs. The use of a buffer enables us to loop over the alpha string first so the innermost loop only acts on the CI elements in the same alpha column. The reduced memory stride leads to improved cache reuse and memory page access patterns. The stride is limited by the size of a single alpha column. Since an alpha column can easily fit in L2 cache (or even L1 cache for small cases), temporary locality can be expected if there are enough operations in the same alpha column. Second, an integral driven loop is used to organize the updating of σ_3. The benefit comes from the effective data reuse in the computation of Hamiltonian matrix elements. Because the last term in H_3 involving the alpha orbital array does not depend on beta strings, except for the integral index, this term can be computed and shared by all the beta pairs with the same integral index. In the current implementation the alpha string orbital information is unpacked and computed only once for each beta integral index list, and the contribution to Hamiltonian H_3 is calculated and stored temporarily. The intermediate result can be utilized by all the beta pairs that belong to the current integral index label, so that no further computation over alpha strings is needed. This strategy for reusing data not only amortizes the overhead of string unpacking but also greatly reduces the computational effort for calculating the Hamiltonian matrix elements.

The algorithms for two different σ_3 implementations are outlined below. The original algorithm has a very simple structure but the computation of the innermost loop is very demanding and not cache friendly. Although the second algorithm looks much more complicated, the innermost loop is very simple. Tests suggest that the optimized (second) algorithm is much more efficient than the original (first) one.

A pure string driven algorithm for σ_3'

```
Loop over each beta string $K_\beta$
    Generate excitation string $K_\beta'$ by applying $E_{jj}$,
    Loop over alpha strings,
        Retrieve orbitals array of alpha string
        Calculate $H_3$, do $\sigma_3$ contribution.
```

A combined driven algorithm for σ_3 calculation:

```
Generate the **integral driven beta list** $L_\beta$
    [do the following if buff is full]
    loop over index *indij*
        loop over alpha strings, precompute and store the
        intermediate result for $H_3$
        loop over alpha strings (***string driven***)
            loop over beta pairs with *indij* (***integral driven***)
                compute $H_3$ and update $\sigma_3$.
```

In the computation of σ_5 a buffer array is used to hold the beta double excitation list temporarily, so the string driven loop over alpha strings can be applied first. However, since the computation of the Hamiltonian matrix element here is rather straightforward, there is no need to use the integral driven approach. The innermost loop is optimized by using a string driven beta double excitation list to improve the cache performance. Note that since the stored beta list is retrieved for every alpha string, it is important to keep the array size in a reasonable range so it can fit into cache. In the code presented here, this is accomplished by imposing an upper limit on the size of the buffer array. The storage of the entire double excitation list is not only unnecessary, but also can be very inefficient, even if sufficient memory is available.

The computation of σ_6 requires the combination of alpha and beta single excitation lists. For this reason the buffer array is divided into two parts, *ialoop* and *ibloop*, to hold the generated alpha and beta excitation lists separately. Still, one loops over the generated alpha excitation pairs first, so the inner loop over the beta list will act on the alpha columns defined by the given alpha pair. Temporary data locality can be exploited if the length of the beta list is long enough, because the more operations on the same chunk of data, the more chance there is to find the required data already in cache. Two major efforts are involved: building the Hamiltonian block and updating the σ vector. To accomplish these tasks, the alpha list is generated in an integral driven loop and the beta list in a string driven loop. The alpha excitation pairs with the same integral index are grouped together so they can use the same integrals and intermediate data when combined with the beta list. Because of the reuse of data, the Hamiltonian matrix blocks, although never explicitly generated, can be computed very efficiently. The string driven beta single excitation list is used in the innermost loop so the computation of the matrix vector product can be performed in a cache efficient way. For $Ms=0$, however, the parallel strategy requires the overall structure in the alpha-beta part to be organized in an integral driven approach, so the integral index condition $ij \leqslant kl$ can be applied to halve the computational cost. In such cases, an integral driven beta single excitation list is generated and used to organize the innermost loop.

Algorithm for alpha-beta routine, $Ms \neq 0$:

```
Generate the **integral driven alpha list** $L_\alpha$
    [do the following if buff is full]
    Generate the ***string driven beta list*** $L_\beta$
        [do the following if buff is full]
        Loop over *indij* of $L_\alpha$
            Precompute the intermediate quantities for $H_6$
            Loop over alpha pairs with index *indij* (**integral driven**)
                Loop over beta pairs (**string driven**), do $\sigma_6$ contribution.
```

Algorithm for alpha-beta routine, $Ms=0$:

```
Generate the **integral driven alpha list** $L_\alpha$
    [do the following if buff is full]
    Generate the ***integral driven beta list*** $L_\beta$
        [do the following if buff is full]
        Loop over *indij* of $L_\alpha$
            Loop over alpha pairs with index *indij* (**integral driven**)
                Loop over *indkl* of $L_\beta$, $indkl \leqslant indij$,
                    Loop over beta pairs with index *indkl* (**integral driven**),
                    do $\sigma_6$ contribution.
```

III. PARALLEL IMPLEMENTATION

A. Parallization of subspace operations

In the parallel FCI program presented here both the CI coefficient vectors C and the corresponding σ vectors are kept in distributed memory. The CI vector is first organized in symmetry blocks, and then distributed evenly over all available processors. To avoid explicit access of a single array element, an alpha column is chosen as the smallest unit for a network communication. Therefore the determinants in the same alpha column must be kept on the same node.

The iterative Davidson diagonalization procedure is chosen as the first step because of the consideration of both parallel efficiency and memory capacity. According to Amdahl's law, even a small percentage of serial code can cause serious scalability problems in parallel applications. Moreover, the memory requirement for the huge subspace vectors can generally only be partially solved using distributed memory.

Each Davidson iteration involves several vector operations, the construction of the Hamiltonian matrix in the subspace, the computation of the residual vector, the preconditioning of the residual, Schmidt orthogonalization, and the normalization of the new basis vector. Fortunately, the parallelization of these subspace operations is rather straightforward. Since all the subspace vectors are distributed, the computationally intensive vector operations can be carried out locally on each processor, and only scalar values are required to communicate through network calls. The communication overhead here is negligible.

The most difficult part of the parallelization is the *HC*

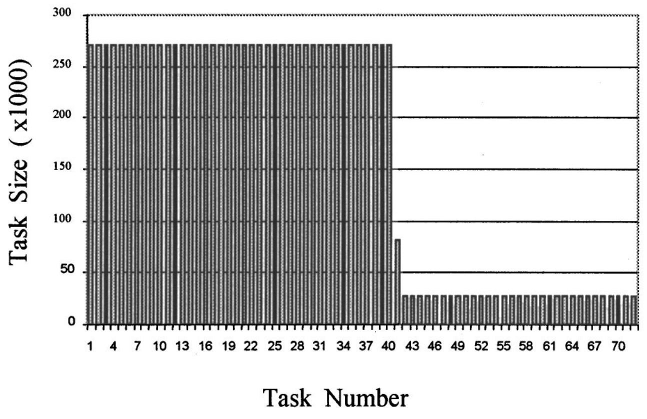

FIG. 1. The size of subtasks in calculation of H_3COH (14,14) using replicated data parallel approach. 16 processors are used. The task parameters are set to $Ntask\_proc=4.5$, $R\_size=10$, $R\_num=1.2$.

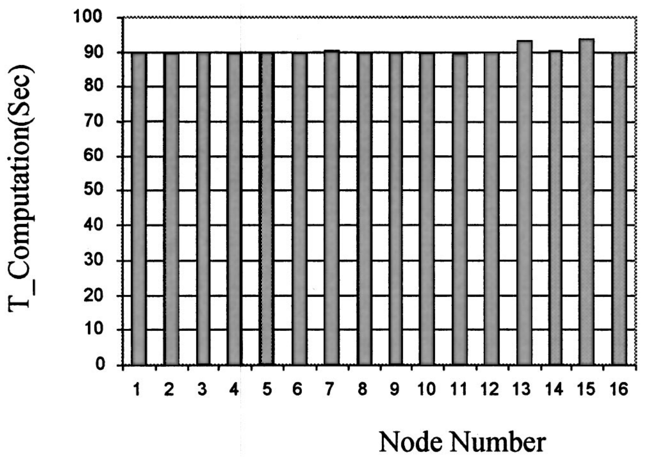

FIG. 2. Load balancing of HC step on 16 node PC cluster. The calculation is for H_3COH (14,14) using the replicated data parallel approach. The parameters are set to $Ntask\_proc=4.5$, $R\_size=10$, $R\_num=1.2$.

step, in which the local σ vector is obtained from basis vector C. Both replicated data and distributed data algorithms have been implemented to parallelize this step.

B. Replicated data implementation

In a replicated data implementation two working arrays X and Y are needed to hold the entire CI vector and the partially formed σ vector. As both the C and σ vectors are originally distributed, a *ddi_allgatherv* (*MPI_allgatherv*) operation is called to collect the distributed vector C into the working array X on each processor. The *HC* calculation can then be executed in parallel and the partial σ vector obtained in array Y. After the *HC* calculation, the partial σ result is summed and scattered into the σ array on each processor by calling the standard *ddi_reduce_scatter* (*MPI_Reduce_scatter*) operation. No communication is involved inside the *HC* computation.

The key issue in parallelizing the *HC* task is to obtain good load balancing on each processor. To achieve this, a dynamic load balancing strategy is employed. The entire *HC* computation is split into many subtasks, each of which is defined as updating a segment of the σ vector. A task list is generated before the *HC* calculation, and the tasks are dynamically assigned to processors at run time.

Good load balance can be easily achieved by generating a long task list and many small tasks. However, the associated communication and computation overhead is undesirable. As a compromise between load balance and the communication overhead, task aggregation is done to artificially produce some large tasks. Three parameters are used in the program to define the parallel subtasks:
(i) *Ntask_proc*: the average number of tasks per processor.
(ii) *R_size*: the ratio of the size of large tasks to that of small tasks.
(iii) *R_num*: the ratio of the number of large tasks to that of small tasks.

The parameters are set to default values and can also be specified by input directives. Using these parameters to define the task list also gives us more flexibility porting the code to different hardware platforms.

Once generated, the tasks are properly arranged in the task list. Good load balance can be obtained by executing the tasks in the order of decreasing size. Figures 1 and 2 give an example of a task list and the resulting load balance on a 16 node PC cluster.

C. Distributed data implementation

In the distributed data implementation, a segment is defined as the part of a CI vector on each node. To make full use of the data locality, a static load balancing scheme is employed. The computational task of each node is defined as updating its local σ segment. Using this task definition, interprocessor communication is only required for collecting the CI coefficient vectors C. The network communication can be carried out in a one-sided asynchronous manner, so that the remote CI coefficient vectors can be easily accessed without the collaboration of remote nodes. The one-sided communication feature is supported by DDI (distributed data interface) routines. In DDI two processes are spawned on each processor. One of these performs the computation; the other becomes the data server. A copy of the local C vector is required in the data server so it can be accessed by other remote processors. In the distributed data implementation the need for the replicated arrays *X and Y* is removed. Instead, a small sized buffer array Z is used to facilitate both communication and computation. The remote CI coefficients, if needed, are first collected in the local buffer Z, and the updating of the σ segment can be carried out locally. The two approaches are illustrated in Figs. 3 and 4.

1. Beta-beta routine

The beta-beta routine involves the single and double excitations of beta strings. Since the CI vector is distributed by alpha columns, this part can be done completely locally. The sequential algorithms can be applied here, but only the local alpha strings should be involved because only the local σ vector will be updated. However, each processor needs to create the entire set of beta single and double excitation lists, so the list generation process is duplicated. This can cause problems in the parallelization effort.[8,17] The most effective

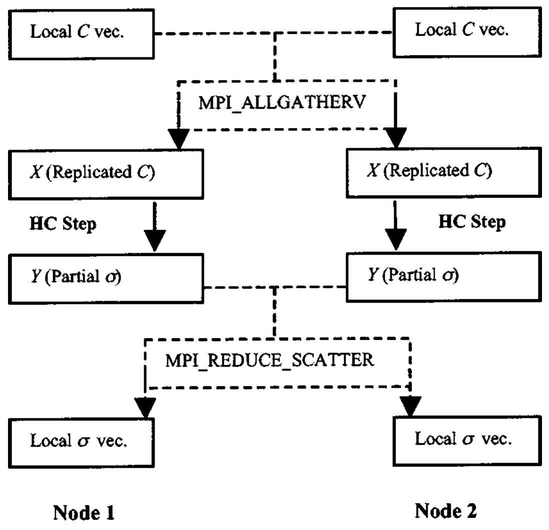

FIG. 3. Outline of replicated data parallel FCI.

way to solve this problem is to decrease the computation cost. In the current implementation the overhead is greatly reduced by an efficient list generation algorithm, so its impact on the overall performance is almost negligible.

2. Alpha-beta routine

The calculation of σ_6 is the most time consuming step. Since the alpha columns are globally distributed, this part also incurs intensive network communication to get the CI coefficients from remote processors. To exploit the sparseness of the FCI Hamiltonian matrix, network vector *gather* and *scatter* operations are commonly used to collect and distribute the remote data. The *gather* operation applies to the C vector, and the *scatter* operation applies to the σ vector. According to the task definition, only the local part of the σ vector is updated, so the *scatter* operation is not necessary in the current implementation.

To effectively hide any latency of the communications subsystem, it is favorable to send a small number of large messages instead of a large number of small messages. For this reason, a buffer array is used to collect and store the remote alpha columns. The computational procedure of the alpha-alpha routine can be divided into three steps. First, the alpha single excitation list L_a is generated by looping over the local alpha strings and applying all the $\hat{E}_{ij}$ operators. Second, the required remote alpha blocks in the excitation list are gathered into the local buffer array via network communication. Third, the beta list L_b is generated and combined with the alpha list, and the σ_6 contribution can be calculated using the coefficient vector gathered in the buffer array. Communication only occurs in the second step; all the other steps are completely local. In order to avoid sending and receiving many small single alpha columns, an optimized communication technique can be applied by grouping the required remote alpha strings according to the nodes on which they reside. After grouping one can send the *gather* request of many alpha columns to the remote node in one communication call. The entire vector gathering step is completed by sending such a request to and receiving the coefficient vector from remote nodes one by one.

Even with such an optimization, the above algorithm can still be very communication intensive. The *gather* step mentioned above is a collective operation; thus it requires the completion of all work on the participating nodes. Such an operation is not efficient on clusters. Especially if it is done in block mode, the communication must be carried out one by one and in a predefined order, which adds additional synchronization overhead. Moreover, there is the problem of "redundant communication." Since the size of buffer array is always limited, the whole process of list generation generally requires many loop cycles to complete, thus the same remote alpha string may be generated and requested many times in different loop cycles. As shown in Fig. 5, the same remote alpha blocks may be fetched through network communication repeatedly. Although such problems have not been addressed by other distributed data FCI implementations, it is important to decrease communications to a minimum in order to achieve good performance on clusters.

The solution to this problem in the algorithm presented here is to replace the redundant communication with extra computation; trading cycles for communication. Instead of looping over only the local alpha strings, one loops over the entire alpha strings. The entire alpha string space is swept and the remote alpha string is placed into the gather list only if it has a single excitation string that is local. Since each alpha string is looped only once, redundant communication is avoided. At the same time the collective *gather* is decoupled into point-to-point *gather* automatically if one computes the contribution of σ_6 node by node, as illustrated in Fig. 5. Note that repeated computation occurs here since the entire alpha single excitation list is generated and checked on each node. However, because the string driven list in the current algorithm can be generated very efficiently, such extra computation has little impact on the overall performance. Another important issue is the network contention that might arise if all nodes start the loop from the first node. This situation can be partially eased by changing the starting node of each computational processor. The starting node can be

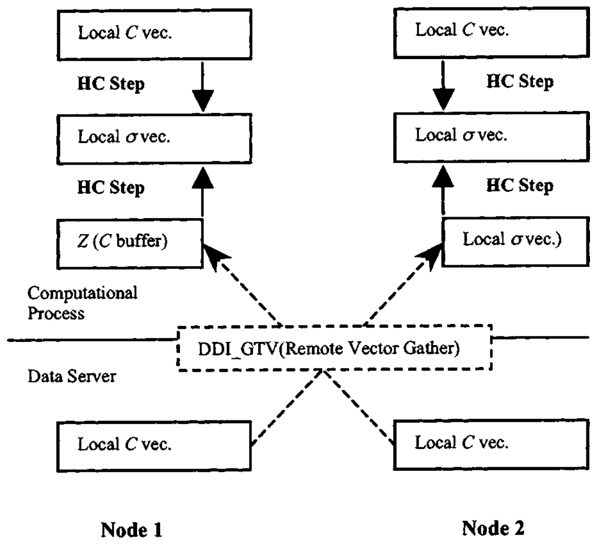

FIG. 4. Outline of distributed data parallel FCI.

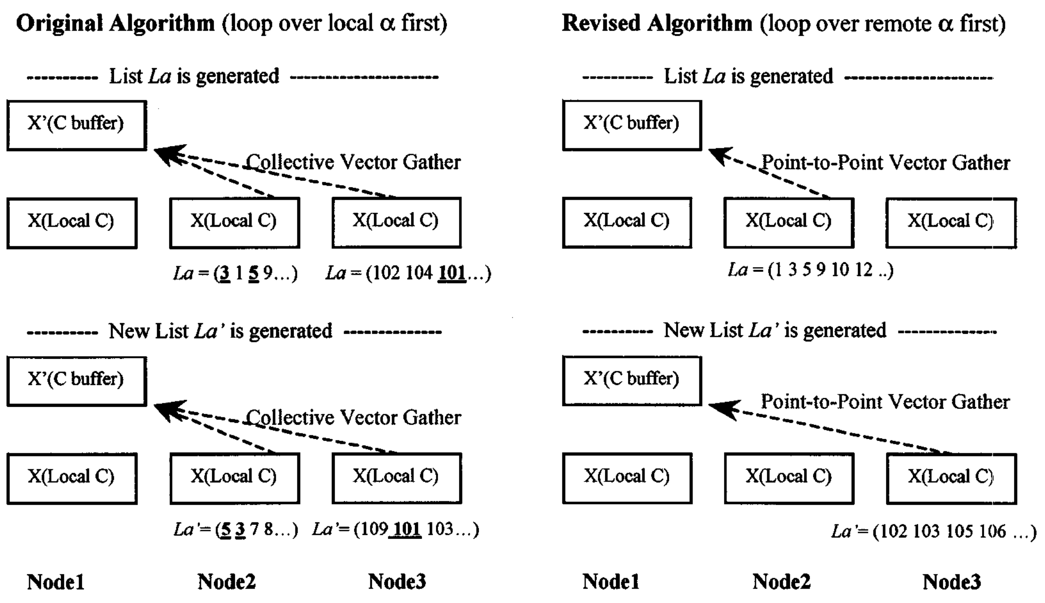
Original Algorithm (loop over local α first)
Revised Algorithm (loop over remote α first)
---------- List La is generated --------------------
X'(C buffer)
Collective Vector Gather
Point-to-Point Vector Gather
X(Local C)
La = (3 1 5 9…) La = (102 104 101…)
La = (1 3 5 9 10 12 ..)
---------- New List La' is generated -------------
La'= (5 3 7 8…) La'= (109 101 103…)
La'= (102 103 105 106 …)
Node1
Node2
Node3

FIG. 5. Communication patterns of two different algorithms for alpha-beta routine. Numbers in the graph are for illustration purpose only. The numbers in bold indicate redundant communication.

randomly chosen or can be specified in a predefined order. In this code each node starts from its local alpha strings, and then it loops over the remaining nodes in a cyclic order. In practice better results are obtained in this way, although the difference is not significant.

Once the required CI coefficients are fetched into the buffer array, the σ_6 construction can be carried out completely locally. The sequential algorithm discussed above is applied, and the computation is organized in a combined CI driven approach. For $Ms=0$ the innermost loop must be organized in an integral driven approach, because of the static load balancing scheme employed here.

3. Alpha-alpha routine

The alpha-alpha routine computes the contribution of σ_2 and σ_4. As discussed above, this can be omitted by applying a vector transposition step for $Ms=0$. Since the calculation of σ_2 involves only single alpha replacement, this part is integrated into the alpha-beta routine and the contribution of σ_2 is calculated as a byproduct of σ_6. This eliminates the communication for σ_2. The term σ_4 involves double excitations of alpha strings and it is treated separately. Intensive network communication occurs here in order to get the remote alpha blocks for the calculation of σ_4. The communication is handled in the same way as in the alpha-beta routine, so repeated communication is avoided.

4. DDI in GAMESS

The distributed data interface[10] (DDI) is the built in facility in GAMESS for network communications. Its underlying libraries include TCP/IP sockets code, MPI-1 and SHMEM libraries. DDI provides basic network communication functions, including point-to-point operations like *send* and *receive* and collective operations like *broadcast* and *global sum*. In addition, DDI also provides features including distributed and globally addressed array and one-sided communication, which are very useful for many quantum chemistry codes.

On clusters, the one-sided communication is implemented by creating two processes on each node. One process handles real computational tasks, and the other becomes the data server. The data server holds the distributed array; its task is to listen to network messages and respond to the data request. Three basic operations are provided in DDI: remote *get*, *put*, and *accumulate*. There are advantages to using one-sided communications. First, it reduces programming difficulty. Second, it is efficient because the asynchronous communication model eases the overhead of explicit synchronization between processors.

However, many applications may require more complicated communication patterns. For example, the *gather* operation used in the σ vector updating step requires noncontiguous access to the distributed data, which is beyond the ability of remote *get*. To use the *get* function the required data must be placed contiguously in the distributed array. One can of course complete the remote *gather* task by repeatedly requesting single alpha columns, but the associated synchronization and latency overhead would be very costly, especially on clusters. In addition, bandwidth is also severely limited if the size of a single alpha column is very small. Clearly, such problems can be solved if the data server is "smart" enough to do a local vector *gather* before sending the data. Such an operation would be very simple if one had an appropriate communication library interface.

To add more functionality to DDI, a message handling function *ddi_user* was created, and a DDI message ID from 99 to 199 is reserved for it. Inside *ddi_user*, the programmer can assign a reserved ID to a unique communication pattern and develop the corresponding request handling function. The next step is to pass the message and control from DDI to *ddi_user*. The DDI *data server* is a daemon process that

TABLE II. Timing comparison of calculations on single CPU IBM Power3II 375-MHz node.

Molecule	Point group	Irrep	Spin mult	Nel	Norb	Dimension (determ)	Time per iteration(s) Orig. FCI	Time per iteration(s) DDI FCI	Speed up
H_3COH	C_1	a_1	1	14	14	11 778 624	694.4	352.9	1.97
H_3CSiH_2	C_1	a_1	2	13	14	10 306 296	3734.9	541.6	6.90
HNO	C_S	a_1	1	12	14	4 513 905	151.3	70.5	2.15
H3CN	C_S	a_2	3	12	16	24 986 830	8850.0	986.6	8.98
H_2O	C_{2v}	a_1	1	10	14	1 002 708	25.5	8.1	3.15
C_2H_4	D_{2h}	a_{1g}	1	12	16	8 018 480	281.3	66.2	4.25
Si_2H_4+	D_{2h}	b_{3u}	2	11	16	4 369 587	337.0	42.7	7.89

exists until the program finishes, so a call is placed to *ddi_user* in the daemon loop. The control will be returned to DDI daemon immediately if the message ID does not match those reserved. Otherwise, *ddi_user* will hand the message request to the proper user defined functions via a message mapping mechanism. A remote *gather* operation is handled in the following way. First, the computation process (caller) sends a vector *gather* request to the DDI server. The message is received by the DDI daemon and then passed to *ddi_user*. Since the message ID for *gather* is a reserved one, *ddi_user* will take it and pass the execution control to the target function *ddi_vgather*, where the vector *gather* request is actually handled. Inside *ddi_vgather* the server process first requests and receives the information about the requested alpha lists from the caller, then performs a local *gather* using a buffer array and sends the compiled data to the caller.

The implementation of remote vector *gather* is not a new idea. In fact, ARMCI (Ref. 18) (aggregate remote memory copy interface) also provides a set of descriptive communication interfaces, where a vector version of the *get* operation is provided. However, it is not simple to include all the communication patterns in real applications. With the ability to add user predefined communication routines, the DDI *data server* evolves to a *communication server*, thereby providing more flexibility in developing performance network communication functions. For example, in the vector transposition step required by the simplification condition for $Ms=0$, the DDI server acts in dual roles: it collects the transposed vector blocks and records the property of those blocks. After synchronization, it sends the recorded property array to a local computational process, and then sends the transposed blocks one by one. Such a communication intensive operation generally does not scale well on clusters. However, with the "smart" DDI *communication server* it performs very efficiently and therefore does not create too much overhead.

IV. PARALLEL PERFORMANCE AND ANALYSIS

First, consider the sequential performance of the new parallel code. This is often ignored in parallel applications, but it provides an important baseline against which to compare scalable performance. Further, it is impossible to achieve high performance computing without a good underlying serial code. An inefficient sequential algorithm can cost precious parallel CPU cycles, and this inefficient use of computational resources only increases as more processors are used. In Table II the timing of the *HC* construction per CI iteration is listed for both the original code[9] and the present distributed data parallel code running on one processor. Note that in the sequential code only the upper triangle of Hamiltonian matrix elements is computed. The distributed data code is designed with the concept of local and remote data in mind. When a Hamiltonian matrix element is found only the local σ vector is updated, thus the computation of the entire Hamiltonian matrix is required. Despite the extra computation, the sequential performance is significantly improved. The increased efficiency is even greater in high-spin and high-symmetry cases. The good performance in high-spin cases can be explained by the string driven innermost loop in the alpha-beta routine. For $Ms=0$ an integral driven innermost loop over beta pairs is employed in the parallel code. The string driven innermost loop is more effective in exploiting the spatial locality so better cache and CPU performance can be obtained. In high-symmetry cases the significant performance gain indicates that the list generation algorithm is not only efficient, but also well suited for Abelian group symmetry.

Two example calculations, one singlet and one doublet, are chosen to test the performance and scalability of the new parallel FCI code. The calculations were performed on a PC cluster consisting of 16 PII 400-MHz processors linked to a central hub using a 100-Mbps fast Ethernet connection. The cluster has 8-GB aggregate memory with 512 MB each. The

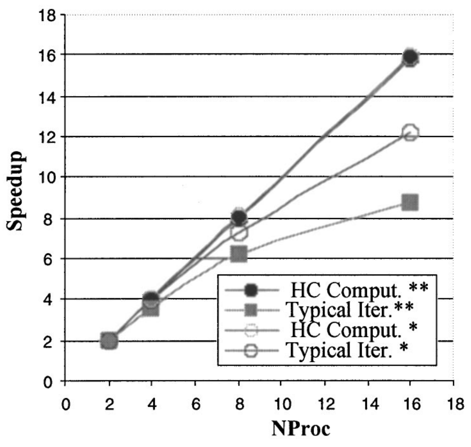

FIG. 6. Parallel performance of replicated data FCI code on PC cluster (PII 400 MHz, 100-Mbps fast Ethernet, 16 nodes).* Doublet state of H_3CSiH_2, 10 306 296 Dets. generated by distributing 13 electrons among 14 active orbitals.** Singlet state of H_3COH, 11 778 624 Dets. generated by distributing 14 electrons among 14 active orbitals.

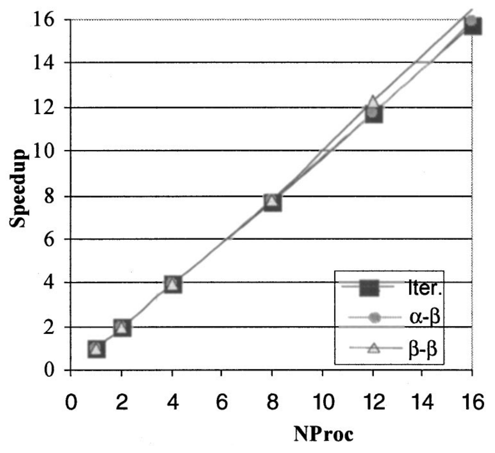

FIG. 7. Parallel performance of distributed data code in FCI (14,14) calculation of singlet state of H_3COH on PC cluster. 11 778 624 Dets are generated by distributing 14 electrons among 14 active orbitals.

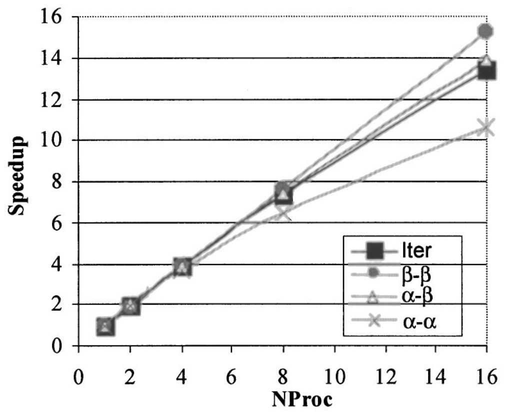

FIG. 8. Parallel performance of distributed data code in FCI (13,14) calculation of doublet state of H_3CSiH_2 on PC cluster. 10 306 296 Determinants are generated by distributing 13 electrons among 14 active orbitals.

singlet calculation on H_3COH uses a FCI space of 11 778 624 determinants generated by distributing 14 electrons over 14 active orbitals. The doublet calculation on H_3CSiH_2 has a 10 306 296 determinant space generated by distributing 13 electrons over 14 active orbitals. In the singlet calculation only beta-beta and alpha-beta routines are performed and the alpha-alpha routine is replaced by a vector (matrix) transposition step; in the doublet calculations the alpha-alpha routine is also executed.

Figure 6 illustrates the parallel performance of the replicated data code. The timing of the *HC* computation as well as that of one typical Davidson iteration is measured and analyzed. In the replicated data approach two collective operations have to be performed before and after the *HC* computation. The standard MPICH library is used for these collective communications. It is shown that in both calculations the computation time of the *HC* step scales almost perfectly, illustrating the effectiveness of the dynamic load balancing scheme. The overall speedup of the doublet calculation is about 12 on 16 nodes, which is reasonably good performance considering the cluster's fast Ethernet connection. However, the speedup of the singlet calculation is only 8.4. Clearly, the ratio of computation to communication is an important factor for parallel efficiency, as in the singlet calculation the computational effort is nearly halved, while the communication remains almost the same. The communication time in the H_3COH calculation takes 47.4% of the total iteration time running on 16 nodes, almost as much as the cost of computation. Of course, one expects the performance to improve using a better MPI library or hand-tuned collective routines. However, it is clear that the collective operations will remain the bottleneck on PC clusters. Besides, the replicated data approach is restricted by the memory capacity of a single CPU node.

The parallel efficiency of the distributed data parallel FCI code was tested on the same PC cluster. The results for the singlet and doublet calculations are presented in Figs. 7 and 8, respectively. The singlet calculation scales almost perfectly; the speedup on 16 nodes is 15.7. A slight super linear speedup is obtained in the beta-beta routine because of the cache effect. Note that in the beta-beta routine the generation of the entire single and double excitation lists is repeated on each node. This is hard to avoid if one is to exploit the data locality, as discussed with regard to the direct list parallel implementation.[8,17] However, they were unable to find a more efficient approach within the framework of an integral driven list generation scheme, so the scalability of the beta-beta routine is severely limited. In the present beta-beta implementation the list generation cost is decreased using the optimized string driven algorithm. The excellent scalability of this beta-beta routine also illustrates that an efficient sequential algorithm can help the parallel performance. Similarly, the repeated list computation in the alpha-beta routine should not present a big problem, as only the single excitation list is involved. Although the overall performance benefits a little from the super linear speedup in the beta-beta routine, the key issue here is still the reduced communication cost. For example, on 16 nodes the time per CI iteration is about 149 sec, and the time for communication is less than 5 sec.

In the doublet calculation the beta-beta routine again shows almost perfect scalability. The speedup of the alpha-beta routine drops a little compared to the singlet calculation but it is still reasonable. Since the alpha-beta routine here is organized in the combined driven approach, its actual performance is better than that in the singlet calculation. The alpha-alpha routine is dominated by communication, so it does not perform as well as the other routines. Two possibilities are under investigation for future improvement of this part. The first method is to combine the alpha-beta and alpha-alpha routine together so the gathered remote vectors can be shared. Because the double excitation list is usually very large, the network performance may be further improved by dynamically choosing between *vector gather* and *block get* based on the volume of communication. Another method is to transpose the CI vector so one can apply the algorithm of the beta-beta routine instead. In this way two vector transposition operations are required, one for the C vector and one for the σ vector. However, the storage of the transposed vectors may increase the memory requirement greatly, unless disk I/O is involved.

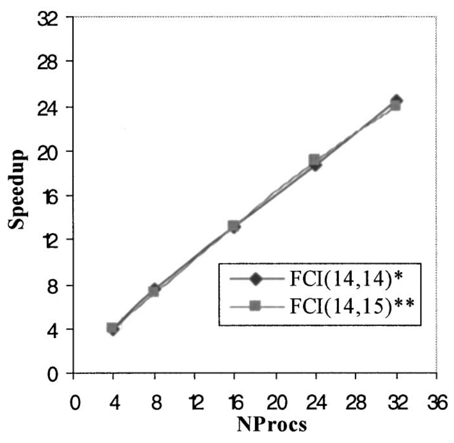

FIG. 9. Parallel performance of distributed data FCI on IBM cluster.* Singlet state of H_3COH, 11 778 624 Dets generated by distributing 14 electrons among 14 active orbitals.** Singlet state of H_2O_2, 41 409 225 Dets generated by distributing 14 electrons among 15 active orbitals.

TABLE III. Timing per iteration of distributed data FCI on IBM cluster. The number of determinants in the CI calculation is 11 778 624 for H_3COH and 41 409 225 for H_2O_2, respectively.

	Time per iteration	
NProc	H_3COH (14,14)	H_2O_2 (14,15)
4	108.63	515.11
8	57.9	287.91
16	33.15	157.64
24	23.15	108.18
32	17.75	86.1

All electron FCI calculations on the potential curve of ground state N_2 have also been performed as a benchmark on the 16 node PII cluster. A small basis set 6-31G was used, and D_{2h} symmetry employed. A total of 16 different geometries were computed to obtain the potential curve. The FCI (14,18) calculation includes 126 608 256 determinants in the CI space, and it requires 6.3 GB aggregate memory, close to the cluster's 8-GB memory limitation. The N_2 calculation takes 750 sec per CI iteration, of which about 85 sec are spent on network communication and about 20 sec are lost due to load imbalance. There is a total of 12.6 GB data transferred per iteration and the average network bandwidth is about 75 Mbps, showing the effectiveness of the improved DDI communication server. The efficiency of CPU usage is measured as 83% for the entire calculation.

The parallel performance of the distributed data parallel code was also tested on an IBM pSeries p640 cluster. The IBM cluster represents the high end hardware platform in cluster computing. Each IBM node consists of 4 power3II processors running at 375 MHz, and each node has a dual Gigabit Ethernet 64-bit PCI connection. In contrast with the Pentium II processor, the IBM processor has a much larger 4-MB level 2 cache. It is necessary to keep the size of the buffer array basically unchanged in one set of calculations in order to factor out the cache effect. Otherwise the scalability obtained on the IBM system would be misleading. The calculations were performed on H_3COH (14,14) and H_2O_2 (14,15) using up to 32 processors. As shown in Fig. 9, almost the same linear scalability is obtained in both calculations. Besides the speedup performance, the timing data in Table II is redone in Table III. Despite the slower CPU clock rate, the FCI (14,14) calculation running on the IBM cluster is ~4–5 times faster than that on the PII cluster. The calculation requires only 33 sec per iteration on 16 IBM processors compared with 149 sec on the PC cluster. It is very encouraging that one iteration of a FCI (14,14) calculation can be completed within merely 18 sec. It is therefore expected that the parallel algorithm presented here will considerably expand the potential applicability of CASSCF, making calculations with such large or even larger active spaces routine. In Ref. 6 a similar CASSCF (14,14) calculation was performed on an IBM SP2 using eight four-way Power3 nodes; one CI iteration took about 270 sec to complete. The two calculations are comparable, as in the FCI only the active orbitals are involved and C_I symmetry was used in both calculations. The present calculations were executed on Power3II 375-MHz nodes, which are twice as fast as the Power3 200-MHz nodes. The FCI (14,15) calculation on H_2O_2 with 41 million determinant basis was also performed to examine how the efficiency changes with the size of the problem. As illustrated in Fig. 9, the increase in execution time is proportional to the increase in the CI dimension, indicating good efficiency of the algorithm in large calculations.

V. CONCLUSIONS

The implementation of both replicated and distributed data parallel FCI programs is presented in this paper. Data reuse, cache performance, and communication cost are the primary concerns in this parallel implementation. A combined CI driven scheme is implemented to meet these combined requirements. In the CI driven strategy the communication is controlled by a string driven approach, the Hamiltonian matrix is computed in integral driven scheme, and the matrix vector multiplication is again performed using a string driven approach.

Improved string storage and CI address calculation schemes are proposed and implemented. Both string driven and integral driven lists are employed, and their generation algorithms are discussed. The list generation procedure is further combined with a buffer algorithm to balance the storage overhead and the computation cost of list generation.

In the replicated data parallel implementation load balancing is the essential concept. The dynamical load balancing strategy, although quite simple, effectively achieves good work balance and reduces the associated overhead. However, the parallel performance of the replicated data code is still limited by the collective network operations, and the capacity of the program is restricted by the amount of memory available on a single node.

In the distributed data parallel implementation the communication cost is our primary concern. To make better use of data locality a static load-balancing scheme is employed. The communication costs in parallel FCI were analyzed, allowing the redundant communication and its collective operation nature to be identified. With the help of the combined CI driven scheme and an efficient list generation algorithm, it was possible to replace redundant communication with inexpensive computation, and decouple the collective *gather* into point-to-point *gather* at the same time. In addition, an optimized *gather* routine was implemented in DDI, and the network performance is improved correspondingly with reduced latency and synchronization overhead.

The parallel codes were tested on both a low end PC cluster and a high end IBM cluster. The sequential performance is significantly improved, and the parallel code exhibits linear parallel scalability on both platforms. The example calculations show that the new distributed data parallel algorithm could extend the supported hardware for parallel FCI calculations to even low end PC clusters, and the high computational performance archived on IBM clusters shows its great potential in expanding the applicability of CASSCF applications.

Several issues are worthy of further investigation, including the cache effect on the buffer arrays, load balance in the distributed data scheme, optimization of the alpha-alpha routine, out of core storage of the subspace vectors, and implementations that make use of the SHMEM library.

ACKNOWLEDGMENTS

We are grateful to Professor Wen, Professor Ruedenberg, Dr. Joe Ivanic, and Dr. Mike Schmidt for helpful discussions. This research was performed in the Ames Laboratory Scalable Computing Laboratory (SCL), and sponsored by the Mathematical, Information and Computational Sciences Division of the Office of Advanced Scientific Computing Research, Office of Science of the U.S. Department of Energy under Contract No. W-7405-eng-82 and the DOE SciDAC program. The Linux PC cluster *spider* was the platform for our code development and testing work. It was funded by the Fundamental Interactions program of the Ames Laboratory, supported by the U.S. DOE Basic Energy Sciences Division. Part of the calculations in this work were performed on an IBM workstation cluster made possible by grants from IBM in the form of a Shared University Research grant, the United States Department of Energy, and the United States Air Force Office of Scientific Research.

[1] P. E. M. Siegbahn, Chem. Phys. Lett. **109**, 417 (1984).
[2] N. C. Handy, Chem. Phys. Lett. **74**, 280 (1980).
[3] J. Olsen, B. O. Roos, P. Jorgensen, and H. J. A. Jensen, J. Chem. Phys. **89**, 2185 (1988).
[4] S. Zarrabian, C. R. Sarma, and J. Paldus, Chem. Phys. Lett. **155**, 183 (1989).
[5] M. Klene, M. A. Robb, M. J. Frisch, and P. Celani, J. Chem. Phys. **113**, 5653 (2000).
[6] E. Rossi, G. L. Bendazzoli, and S. Evangelisti, J. Comput. Chem. **19**, 658 (1998).
[7] C. D. Sherrill and H. F. Schaefer III, Adv. Quantum Chem. **34**, 143 (1999).
[8] R. Ansaloni, G. L. Bendazzoli, S. Evangelisti, and E. Rossi, Comput. Phys. Commun. **128**, 496 (2000).
[9] J. Ivanic and K. Ruedenberg, Theor. Chem. Acc. **106**, 339 (2001).
[10] M. W. Schmidt, K. K. Baldridge, J. A. Boatz *et al.*, J. Comput. Chem. **14**, 1347 (1993); G. D. Fletcher, M. W. Schmidt, and M. S. Gordon, Adv. Chem. Phys. **110**, 267 (1999); Comput. Phys. Commun. **128**, 190 (2000).
[11] E. R. Davidson, J. Comput. Phys. **17**, 87 (1975).
[12] B. Liu, *Numerical Algorithms in Chemistry: Algebraic Methods*, edited by C. Moler and I. Shavitt, LBL-8158 Technical Report, Lawrence Berkeley Laboratory, Berkeley, CA, 1978.
[13] Z. Gan, K. Su, Y. Wang, and Z. Wen, Sci. China, Ser. B: Chem. **42**, 43 (1999).
[14] Y. Wang, Z. Gan, K. Su, and Z. Wen, Sci. China, Ser. B: Chem. **42**, 649 (1999).
[15] P. J. Knowles and N. C. Handy, Chem. Phys. Lett. **111**, 315 (1984).
[16] Z. Gan, Y. Wang, K. Su, and Z. Wen, J. Comput. Chem. **22**, 560 (2001).
[17] L. Gagliardi, G. L. Bendazzoli, and S. Evangelisti, J. Comput. Chem. **18**, 1329 (1997).
[18] J. Nieplocha and B. Carpenter, *ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems*, Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP'99, San Juan, Puerto Rico, April 1999, in (1) Parallel and Distributed Processing, edited by J. Rolim *et al.*, Springer-Verlag LNCS (1986), and (2) IPPS/SDP'99 CDROM (1999).