Parallel implementation of many-body mean-field equations

C. R. Chinn[1], A. S. Umar,[1,2] M. Vallières,[3] and M. R. Strayer[2]

[1] *Department of Physics and Astronomy, Vanderbilt University, Nashville, Tennessee 37235*

[2] *Center for Computationally Intensive Physics, Physics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831-6373*

[3] *Department of Physics and Atmospheric Sciences, Drexel University, Philadelphia, Pennsylvania 19104*

(Received 6 June 1994)

We describe the numerical methods used to solve the system of stiff, nonlinear partial differential equations resulting from the Hartree-Fock description of many-particle quantum systems, as applied to the structure of the nucleus. The solutions are performed on a three-dimensional Cartesian lattice. Discretization is achieved through the lattice basis-spline collocation method, in which quantum-state vectors and coordinate-space operators are expressed in terms of basis-spline functions on a spatial lattice. All numerical procedures reduce to a series of matrix-vector multiplications and other elementary operations, which we perform on a number of different computing architectures, including the Intel Paragon and the Intel iPSC/860 hypercube. Parallelization is achieved through a combination of mechanisms employing the Gram-Schmidt procedure, broadcasts, global operations, and domain decomposition of state vectors. We discuss the approach to the problems of limited node memory and node-to-node communication overhead inherent in using distributed-memory, multiple-instruction, multiple-data stream parallel computers. An algorithm was developed to reduce the communication overhead by pipelining some of the message passing procedures.

PACS number(s): 02.70.Jn, 02.70.Rw

I. INTRODUCTION

The mean-field formalism has been a successful approach to fundamental studies of atomic and nuclear many-body problems [1–4] and promises to be a crucial element in the future development of nuclear astrophysics. The developed technology is directly applicable to Hartree-Fock codes used in a number of fields and could provide a more detailed understanding of such diverse phenomena as molecular dynamics [5, 6] and assist in the design of new biomolecules. The computational techniques are also applicable to many diverse fields, including smooth particle hydrodynamics and nonlinear stiff equations on parallel supercomputers.

From the numerical standpoint, new techniques have been developed to handle the solution of the Hartree-Fock equations on a space-time lattice. In this case these techniques are applied to the nuclear many-body problem. In particular, equations of motion were obtained via the variation of the lattice representations of the constants of the motion, such as the total energy [7–10]. In this variation after discretization approach, the resulting equations exactly preserve the constants of the motion. The lattice techniques are important because the alternative basis expansion approach requires the optimization of basis set parameters due to the finite cutoff in the number of basis states. This procedure becomes very inefficient for large scale calculation of, say, multidimensional energy surfaces. Due to their extensive computational requirements most numerical calculations have employed low-order, finite-difference discretization techniques. For example, calculations were done with the assumptions of spherical or cylindrical symmetry, with no spin degrees of freedom, or z-parity symmetry. With the advent of new supercomputer technologies, it has become feasible to carry out more extensive Hartree-Fock studies without resorting to the symmetry assumptions employed in the earlier applications. These calculations are very large and require Grand Challenge level computing resources.

In order to be able to perform such calculations and to obtain a more detailed comparison with data it is necessary to exploit higher-order interpolation techniques. This is due to the fact that precise studies require overall accuracies at the level of one part in 10^3. Considering the fact that such small numbers arise from the cancellation of large negative and positive parts, the calculational accuracy of each part needs to be better than one part in 10^7. Discretization of the energy functional on a spline collocation lattice provides a highly accurate alternative to the finite-difference method [11]. One significant advantage of this technique is that in comparison to the finite-difference method the same level of accuracy can be attained with a smaller number of lattice points. The basis-spline collocation method has been recently applied to the study of low-energy structure and reactions [12, 13], to the solution of the relativistic time-dependent Dirac equation in strong electromagnetic fields [14], and to the solution of the relativistic hydrodynamics equations [15]. In these applications, the implementation of the basis-spline collocation method has allowed for very accurate calculations of various quantities while using relatively coarse meshes. Nonuniform grids may also be employed with facility in this method. An extensive discussion of the mathematical and numerical properties of splines can be found in several references [11]. The structure of the resulting lattice representation is highly suited

for vector and parallel supercomputers and the method allows a highly modular programming where the order of the splines can be defined as an input parameter.

In the next section we briefly outline the basic equations that are to be studied, introduce the general features of the basis-spline collocation method, and discuss the discretization of the equations, as well as the preconditioned iteration algorithm and the Gram-Schmidt orthogonalization process as a part of this iteration scheme. In Sec. IV we discuss the parallel implementation of the program. Section V will outline the timing studies on the parallel computers Intel iPSC/860 and Paragon, Cray 2, and IBM RS/6000. The paper ends with the discussion of the results.

II. FORMALISM AND NUMERICAL DISCRETIZATION

A. Continuous equations

The details of the derivation of the Hartree-Fock equations can be found in [7–10]. The result for a many-body Hamiltonian containing a one-body kinetic energy and two- and three-body momentum-dependent potential terms is a coupled set of nonlinear partial differential eigenvalue equations,

$$\mathbf{h}\chi_\alpha = \epsilon_\alpha \chi_\alpha \,, \tag{1}$$

where χ_α is a two-component vector (spinor)

$$\chi_\alpha = \begin{pmatrix} \chi_\alpha^+ \\ \chi_\alpha^- \end{pmatrix} . \tag{2}$$

The Hamiltonian $\mathbf{h}$ has the following form (using natural units $\hbar = 1$, $c = 1$, $m = 1$):

$$\begin{aligned} &\mathbf{h} = -\tfrac{1}{2}\nabla^2 + W(\rho,\tau,\mathbf{j},\mathbf{J}) \,, \\ &W = V_N(\mathbf{r}) + V_C(\mathbf{r}) \,, \end{aligned} \tag{3}$$

where V_N is the nuclear potential depending on various currents and densities and V_C is the Coulomb interaction. The densities and currents depend on the states χ_α and are explicitly given by

$$\rho(\mathbf{r}) = \sum_\alpha w_\alpha \{|\chi_\alpha^+(\mathbf{r})|^2 + |\chi_\alpha^-(\mathbf{r})|^2\} \,, \tag{4}$$

$$\tau(\mathbf{r}) = \sum_\alpha w_\alpha \{|\nabla\chi_\alpha^+(\mathbf{r})|^2 + |\nabla\chi_\alpha^-(\mathbf{r})|^2\} \,, \tag{5}$$

$$\begin{aligned} \mathbf{j}(\mathbf{r}) = \sum_\alpha w_\alpha \{ &\mathrm{Im}[\chi_\alpha^{+*}(\mathbf{r})\nabla\chi_\alpha^+(\mathbf{r}) \\ &+\chi_\alpha^{-*}(\mathbf{r})\nabla\chi_\alpha^-(\mathbf{r})]\} \,, \end{aligned} \tag{6}$$

$$\mathbf{J}(\mathbf{r}) = -i \sum_{\substack{\alpha \\ \mu\mu'=\pm}} w_\alpha \chi_\alpha^{\mu *}(\mathbf{r})(\nabla\times\sigma)\chi_\alpha^{\mu'}(\mathbf{r}') \,. \tag{7}$$

V_C requires the solution of the Poisson equation in three-dimensional geometry,

$$\nabla^2 V_C(\mathbf{r}) = -4\pi e^2 \rho(\mathbf{r}) \,. \tag{8}$$

As can be seen from above the solution of the system of equations (1) has to be obtained self-consistently and an accurate solution requires a good representation of various derivatives of the states χ_α. Currently, most Hartree-Fock (HF) and time-dependent Hartree-Fock (TDHF) calculations are performed using finite-difference lattice techniques. It is desirable to investigate higher-order interpolation methods which result in the improvement of the overall accuracy and reduction in the total number of lattice points. The lattice solution of differential equations on a discretized mesh of independent variables may be viewed to proceed in two steps. (1) Obtain a discrete representation of the functions and operators on the lattice. (2) Solve the resulting lattice equations using iterative techniques. Step (1) is an interpolation problem for which we could take advantage of the techniques developed using the spline functions [16, 11]. The use of the spline collocation method leads to a matrix-vector representation on the collocation lattice with a metric describing the transformation properties of the collocation lattice.

B. Splines

Given a set of points or *knots* denoted by the set $\{x_i\}$, a spline function of order M, denoted by B_i^M, is constructed from continuous piecewise polynomials of order $M-1$. These splines have continuous derivatives up to an $(M-2)$nd derivative and a discontinuous $(M-1)$st derivative. We only consider odd-order splines or even-order polynomials for reasons related to the choice of the collocation points. The ith spline is nonzero only in the interval (x_i, x_{i+M}). This property is commonly referred to as limited support. The knots are the points where polynomials making up the spline join. In the interval containing the tail region, the splines fall off very rapidly to zero. The explicit construction of the splines is explained elsewhere [11]. We can also construct exact derivatives of splines provided the derivative order does not exceed $M-1$.

A continuous function $f(x)$, defined in the interval $(x_{\min}, x_{\max})$, can be expanded in terms of spline functions as

$$f(x) = \sum_i B_i^M(x) c^i \,, \tag{9}$$

where quantities c^i denote the expansion coefficients. We can solve for the expansion coefficients in terms of a given, or to be determined, set of function values evaluated at a set of data points more commonly known as *collocation points*. There are a number of ways to choose collocation points [11, 16]; however, for odd-order splines a simple choice is to place one collocation point at the center of each knot interval within the physical boundaries

$$x_\alpha = \frac{x_{\alpha+M-1} + x_{\alpha+M}}{2}, \quad \alpha = 1, \ldots, N. \tag{10}$$

Here, $x_M = x_{\min}$, $x_{N+M} = x_{\max}$, and N is the number of collocation points. Note that collocation points are denoted by greek subscripts. We can now write a linear system of equations by evaluating (9) at these collocation points,

$$f_\alpha = \sum_i B_{\alpha i} c^i \ , \tag{11}$$

where $f_\alpha \equiv f(x_\alpha)$, and $B_{\alpha i} \equiv B_i^M(x_\alpha)$. In order to solve for the expansion coefficients, the matrix **B** needs to be inverted. However, as it stands, the matrix **B** is not a square matrix, since the total number of splines with a nonzero extension in the physical region is $N + M - 1$. In order to perform the inversion, we need to introduce additional linear equations which represent the boundary conditions imposed on $f(x)$ at the two boundary points x_M and x_{M+N}. The essence of the lattice method is to eliminate the expansion coefficients c^i using this inverse matrix. The details of using the boundary conditions and inverting the resulting square matrix are discussed elsewhere [11]. Following the inversion, the coefficients are given by

$$c^i = \sum_\alpha \left[\mathbf{B}^{-1}\right]^{i\alpha} f_\alpha \ . \tag{12}$$

One can trivially show that all local functions will have a local representation in the finite-dimensional collocation space

$$f(x) \longrightarrow f_\alpha \ . \tag{13}$$

The collocation representation of the operators can be obtained by considering the action of an operator $\mathcal{O}$ onto a function $f(x)$

$$\mathcal{O}f(x) = \sum_i [\mathcal{O}B_i^M(x)] c^i \ . \tag{14}$$

If we evaluate the above expression at the collocation points x_α, we can write

$$[\mathcal{O}f]_\alpha = \sum_i [\mathcal{O}B]_{\alpha i} c^i \ . \tag{15}$$

Substituting from Eq. (12) for the coefficients c^i, we obtain

$$\begin{aligned}[\mathcal{O}f]_\alpha &= \sum_{i\beta} [\mathcal{O}B]_{\alpha i} \left[\mathbf{B}^{-1}\right]^{i\beta} f_\beta \\ &= \sum_\beta O_\alpha^\beta f_\beta \ ,\end{aligned} \tag{16}$$

where we have defined the collocation space matrix representation of the operator $\mathcal{O}$ by

$$O_\alpha^\beta = \sum_i [\mathcal{O}B]_{\alpha i} \left[\mathbf{B}^{-1}\right]^{i\beta} \ . \tag{17}$$

Notice that the construction of the collocation space operators can be performed once and for all at the beginning of a calculation, using only the given knot sequence and collocation points. Due to the presence of the inverse in Eq. (17), the matrix O is not sparse. In practice, the operator $\mathcal{O}$ is chosen to be a differential operator such as d/dx or d^2/dx^2. By a similar construction, it is also possible to obtain the appropriate integration weights on the collocation lattice [11].

C. HF equations in collocation space

In order to obtain a set of lattice equations which preserve the conservation laws associated with the continuous equations, it is essential to develop a modified variational approach. This goal is achieved by performing a variation of the discretized form of a conserved quantity, i.e., total energy. Consequently, the resulting equations will preserve all of the conserved quantities on the lattice,

$$\sum_{\alpha\beta\gamma} \Delta V_{\alpha\beta\gamma} \left\{ \mathbf{h}(\alpha\beta\gamma) - \epsilon_\lambda \left| \boldsymbol{\chi}_\lambda(\alpha\beta\gamma) \right|^2 \right\} \ , \tag{18}$$

where the indices α, β, and γ denote the lattice points in three-dimensional space, and $\Delta V_{\alpha\beta\gamma}$ is the corresponding infinitesimal volume element. Due to the presence of derivative operators in the Hamiltonian, the explicit form of these expressions will depend nonlocally on the lattice indices. The general variation, which preserves the properties of the continuous variation, is given by

$$\frac{\delta \boldsymbol{\chi}_\mu^*(\alpha\beta\gamma)}{\delta \boldsymbol{\chi}_\lambda^*(\alpha'\beta'\gamma')} = \frac{1}{\Delta V_{\alpha\beta\gamma}} \delta_{\lambda\mu} \delta_{\alpha'\alpha} \delta_{\beta'\beta} \delta_{\gamma'\gamma} \ . \tag{19}$$

The details of the discrete variation for the finite-difference case are given in Refs. [7, 8]. The three-dimensional expansion in terms of splines is a simple generalization of Eq. (9),

$$\chi_\alpha(x,y,z) = \sum_{ijk} c_\alpha^{ijk} B_i(x) B_j(y) B_k(z) \ . \tag{20}$$

The knots and collocation points for each coordinate can be different. With the appropriate definition of boundary conditions, all of the discretization techniques discussed in the previous section can be generalized to the three-dimensional space. The details of this procedure are given in Ref. [11].

A typical nonlocal term is illustrated below

$$\begin{aligned}(\nabla \chi_\lambda^\pm)_{\alpha\beta\gamma} = &\sum_{\alpha'} D_\alpha^{\alpha'} \chi_\lambda^\pm(\alpha'\beta\gamma)\, \hat{\imath} + \sum_{\beta'} D_\beta^{\beta'} \chi_\lambda^\pm(\alpha\beta'\gamma)\, \hat{\jmath} \\ &+ \sum_{\gamma'} D_\gamma^{\gamma'} \chi_\lambda^\pm(\alpha\beta\gamma')\, \hat{k} \ ,\end{aligned}$$

where the matrices **D** denote the first derivative matrices in x, y, and z directions (they can be different although the notation does not make this obvious) calculated as described in the previous subsection. Finally, the HF equations can be written as matrix-vector equations on the collocation lattice,

$$h\chi_\alpha^\pm \longrightarrow \mathbf{h} \cdot \chi_\alpha^\pm \ . \tag{21}$$

The essence of this construction is that the terms in the single-particle Hamiltonian **h** are matrices in one coordinate and diagonal in others. Therefore, **h** need not be stored as a full matrix, which allows the handling of very large systems directly in memory. The details of this procedure are discussed below.

D. Solution of the discrete HF equations

The solution of the HF equations (21) is found by using the damped relaxation method described in Refs. [17, 4]:

$$\chi_\alpha^{k+1} = \mathcal{O}[\chi_\alpha^k - x_0 \hat{D}(E_0)(\mathbf{h}^k - \epsilon_\alpha^k)\chi_\alpha^k] , \tag{22}$$

where $\mathcal{O}$ stands for Gram-Schmidt orthonormalization. The preconditioning operator $\hat{D}$ is chosen to be [17, 4]

$$\hat{D}(E_0) = \left[1 + \frac{\hat{T}}{E_0}\right]^{-1} \approx \left[1 + \frac{\hat{T}_x}{E_0}\right]^{-1} \left[1 + \frac{\hat{T}_y}{E_0}\right]^{-1} \left[1 + \frac{\hat{T}_z}{E_0}\right]^{-1} ,$$

where $\hat{T}$ denotes the kinetic energy operator. The solution is obtained by an iterative scheme as outlined below.

(1) Guess a set of orthogonal single-particle states.
(2) Compute the densities (4)–(7).
(3) Compute the Hartree-Fock potential.
(4) Solve the Poisson equation.
(5) Perform a damping step (22) without orthogonalization.
(6) Do a Gram-Schmidt orthogonalization of all states.
(7) Repeat, beginning at step 2, until convergence.

In practical calculations we have used the damping scale value $x_0 \approx 0.05$ and the energy cutoff $E_0 \approx 20.0$. As a convergence criterion we have required the fluctuations in energy

$$\Delta E^2 \equiv \sqrt{\langle H^2 \rangle - \langle H \rangle^2} \tag{23}$$

to be less than 10^{-5}. This is a more stringent condition than the simple energy difference between two iterations, which is about 10^{-10} when the fluctuation accuracy is satisfied. The calculation of the HF Hamiltonian also requires the evaluation of the Coulomb contribution given by Eq. (8). Details of solving the Poisson equations using the splines are given in Ref. [11].

III. PARALLEL IMPLEMENTATION

In this section we discuss the details of implementing the lattice representation of the Hartree-Fock equations on the Paragon XP/S 5 and XP/S 35 and Intel iPSC/860 hypercube supercomputers at the Oak Ridge National Laboratory. These machines are distributed memory, multiple-instruction multiple-data (MIMD) computers. The Intel iPSC/860 has 128 nodes with 8 Mbyte of memory per node and a peak rating of 60 Mflops per node leading to a 7.6 Gflop aggregate speed; on the XP/S 5 and XP/S 35 models the peak rating per node is 75 Mflops leading to aggregate speeds of approximately 5 Gflops and 38 Gflops, respectively. Among other differences, the iPSC/860 supercomputer is a hypercube architecture whereas the Paragon is a two-dimensional (2D) mesh. The peak internode communication speed of the iPSC/860 supercomputer is 2.8 Mbyte/sec and of the Paragon supercomputer is 200 Mbyte/sec. The nodes are connected according to the binary interconnection scheme.

As with most parallel implementations we face the problem of limited memory per node and the optimization of the algorithms to minimize the communication among nodes. Since the communication is by far the slowest operation, one would like to have substantial CPU usage on each node in order to minimize the fractional communication overhead. There are two ways to parallelize our Hartree-Fock equations. One way is to distribute a number of single-particle states to each node (Hilbert space decomposition) and have all spatial operations (primarily differentiation) occur locally on each node. In this case the only communication necessary is for the construction of densities and currents, and the Gram-Schmidt orthonormalization of all the states. Alternately, one could keep all of the single-particle states on each node but perform a spatial domain decomposition. This requires the distribution of, at least, two of the three spatial dimensions in Cartesian coordinates. In this case nonlocal operations like differentiation require communication across all nodes, whereas the local densities and currents can be constructed without communication (however, some of the currents do contain differential operators). The load comparison between the two approaches is briefly outlined in Table I. Considering the detailed structure of our potential, which contains many differential operations onto single-particle states and densities, it can be shown that the second method leads to significantly more communication in comparison to the former. Furthermore, due to the excellent accuracies obtained by the basis-spline collocation method, the number of lattice points in each Cartesian dimension is seldom larger than 24, which limits the number of nodes that can be used by spatial domain decomposition. Similarly, the maximum number of nodes that can be used is limited by the number of single-particle states in the single-particle decomposition approach. In contrast, for situations in which the load in the spatial domain is extremely heavy, for example, if the number of lattice points in each Cartesian dimension is greater than 100 and the number of single-particle states is small, then it may be more advantageous to use a domain decomposition. In the subsections below we outline various details of the parallel implementation of our large-scale Hartree-Fock program.

TABLE I. Communication overhead in the two parallelization schemes.

HF step	Domain decomposition	Hilbert space decomposition
densities	Global broadcasts	Global sums
potentials	Nearest Neighbor (heavy)	None
Diff. eq. solutions	Nearest neighbor (heavy)	None
Orthogonalization	Global sums	Global broadcasts (heavy)

A. Initial setup and sequential execution

The main program calls a local subroutine called $open(n_p, me, ihost, mptype, iarch)$, which makes all of the machine-dependent function calls to determine its arguments. The variable n_p is the number of allocated processors, *me* is the node number of each node calling the routine, *ihost* is the node number of the host (generally node 0), *mptype* is the process type, and *iarch* is the machine architecture we use to discern among various parallel platforms. These values are then made available to all subroutines. The variable *iarch* is used in various subroutine arguments which have to call routines with different names based on the machine type. For sequential calculations setup routines have been constructed with the same names as these parallel function calls, which essentially do nothing except to set the node number to 0 and the number of nodes to 1. These subroutines are compiled and linked for sequential operations. This allows for the same modified code to run on sequential machines as well.

B. Reading and distribution of input

The compilation of a parallel program usually takes place on a host, which could be a dedicated computer or actually a node. The communication between the host and the nodes is usually much slower than node-to-node communication and therefore should be minimized. For our setup the host is only used for compilation, issuing the commands to allocate machine resources, and the actual loading of the code to all of the nodes. No other calculations are performed on the host except for the transferring of the printed output. In practice, all nodes contain the same program working on different data. One of the node programs is designated to receive and send input/output from the parallel computer to the host computer. We choose node 0 for this purpose. All of the node programs could remain identical since a common logical *if* statement for the node number can select node 0 for this task. We use a broadcast routine on all nodes which distributes the task of passing the input from node 0 to all other nodes using a subcube broadcast algorithm. The details of this algorithm are discussed later. Since typical startup times are about 50 times larger than the time it takes to transfer a single word of data between two nodes, we have packed all of our real and integer input into two temporary work arrays. The reading of input and packing of data into work arrays are done by node 0. After this all nodes execute a call to the broadcast routine $bcast(iarch, buf, mbytes, 0, mtype)$, where *iarch* chooses the appropriate parallel architecture (Paragon or iPSC/860), *buf* contains the real input, *mbytes* is the length of the message in bytes, the "0" corresponds to the originator node of the broadcast, and *mtype* is an integer tag that is incremented every time an input/output operation is performed (it tags the many messages being sent between nodes and is used in order of arrival or departure). As a result, all nodes obtain the input data packed into the array *buf*. Subsequently, each node unpacks the contents of the temporary arrays into the real variable names used in the code.

C. Distribution of single-particle states

The parallelization of the Hartree-Fock program is done by distributing the single-particle states across a number of nodes. This distribution can be complicated by factors such as load balancing and fractional communication overheads. Since communication is far slower than nodal floating point operation speeds, one has to make sure that a significant amount of CPU power is used by each node before performing a communication bound task. The limiting factors are the number of single-particle states that can be fitted into the memory on each node (which in turn depends on the size of the spatial mesh), the type of states allocated to each node, i.e., neutron or proton or both, the amount of floating point operations done per state (which again depends on the size of the spatial mesh), and finally the number of available nodes. It is, of course, clear that any situation in which all nodes do not have the same number of states will lead to an unbalanced load, i.e., some nodes will be performing while others are waiting. Occasionally this will be the case since it is not always possible to exactly match the number of states with the number of available nodes. One can, for example, attempt to allocate one neutron and one proton state to each node. However, this case is not favored by the Gram-Schmidt orthogonalization process, discussed below, since the orthogonalization takes place only among one type of single-particle states. Therefore, the most optimum allocation appears to be one or more states of the same type on each node. In this case, within the Gram-Schmidt process, the neutrons and protons operate within their own rings. The actual balance between the number of states allocated per node and the communication overhead can only be calculated by actual performance analysis, as we have done below.

The initialization of states on each node is done by generating an indexing scheme. According to this scheme each node initially generates its own initial guesses for the single-particle states. Using the total number of neutron and proton numbers a set of Cartesian oscillator quantum numbers is generated. Each node uses its node number in such a way as to have a distinct set of quantum numbers. Using these quanta the initial states are then generated. On a single-node system this is equivalent to the sequential case. Of course, these quantum numbers are transmitted to node 0 for printing purposes, where by looping over the node numbers information in turn is passed to and collected by node 0.

D. Gram-Schmidt orthonormalization

The Gram-Schmidt procedure used to orthogonalize the single-particle wave functions can be summarized in the following equation:

$$\psi_i'(\vec{x},\tau) = \psi_i(\vec{x},\tau) - \sum_{j=1}^{i-1} \psi_j(\vec{x},\tau) \frac{\langle \psi_j(\tau)|\psi_i(\tau)\rangle}{\langle \psi_j(\tau)|\psi_j(\tau)\rangle} ,$$

$$\psi_i'(\vec{x},\tau) = \psi_i'(\vec{x},\tau)/\sqrt{\langle \psi_i'(\tau)|\psi_i'(\tau)\rangle}, \quad i \geq 2 , \qquad (24)$$

where τ labels the isospin index, distinguishing between the proton and neutron states, and the vector $\vec{x}$ represents the grid collocation lattice in Cartesian coordinates. The matrix element $\langle \psi_j(\tau)|\psi_i(\tau)\rangle$ is defined as

$$\langle \psi_j(\tau)|\psi_i(\tau)\rangle \equiv \int d\vec{x}' \psi_j^\dagger(\vec{x}',\tau)\psi_i(\vec{x}',\tau) . \qquad (25)$$

On a distributed-memory parallel machine difficulties arise since the wave functions are spread out over the nodes of the computer; hence the wave function vectors must be passed between the nodes during the orthogonalization process. This creates a large communications overhead since the wave function vectors are dimensioned by the three-dimensional collocation lattice and are therefore very large ($\sim$ 0.25–0.5 Mbytes). Initially, on each node, the local wave functions ψ are normalized. The sequence of operations is first for node $me = 0$, the local ψ's are orthogonalized with respect to each other via Eq. (24). In many cases time-reversal symmetry is assumed ($itim = 1$), where only half of the total number of states need be explicitly considered. In this case it is still necessary to orthogonalize the wave functions with respect to the time-reversed states.

$$\psi_t(\vec{x},\tau) \equiv \widehat{T}\psi(\vec{x},\tau), \qquad (26)$$

where the operator $\widehat{T}$ is the time-reversal operator. At this time the orthogonalized states are then normalized. Not only are normalized states desired in the end, but for subsequent orthogonalizations the Gram-Schmidt procedure assumes normalized states. Node $me = 0$ broadcasts its local wave vectors to all of the other nodes, which then orthogonalize their local wave vectors with respect to the ones received from node 0, as well as the time-reversed partners. Next node 1 proceeds to orthogonalize all of its local vectors as node 0 has just done. Node 1 then normalizes its local ψ's and broadcasts them to all of the other nodes. For nodes $me > inode$ the local wave functions are orthogonalized with respect to the received wave vector ψ'. Again the time-reversal case can also be considered. The procedure proceeds for all of the nodes, except that it is not necessary for the last node to broadcast the local ψ's. At the end all of the wave functions are already normalized on their respective local nodes.

To optimize the Gram-Schmidt procedure for use on the Paragon supercomputer several modifications were made. One change alluded to earlier is to place the neutron and proton states on nonoverlapping sets of nodes. The ideal situation is to place either one neutron or one proton state only on each node. The Gram-Schmidt procedure would then proceed among the neutron and proton states separately with no communication between the two sets during the orthogonalization process. The neutron states are placed in nodes $0, 1, \ldots, n_p(1)-1$, while the proton states are placed into nodes $n_p(1), n_p(1)+1, \ldots, n_p(1)+n_p(2)-1$.

A second modification involved using the Paragon communication *gsendx* in place of using *bcast*. It is clear that not all of the nodes need to receive the broadcasted wave function vectors, especially with the separate neutron and proton sectors. The routine *gsendx* sends a vector to a specific set of destination nodes, defined in an integer array in a semiglobal operation. The simplest way to do this is to fill an array with all of the destination nodes in either the proton or neutron sector in decreasing node number:

$$ianode(i,1) = n_p(1) - i, \qquad \forall\, i = 1, n_p(1) ,$$
$$ianode(i,2) = n_p(1) + n_p(2) - i, \qquad \forall\, i = 1, n_p(2) ,$$

where $n_p = n_p(1) + n_p(2)$, and then to call *gsendx* with the appropriate number of destination nodes specified.

This algorithm is simple, but inefficient. Because the receiving nodes are listed in decreasing order in *ianode*, the last node in the Gram-Schmidt sequence receives the broadcasted vector first. For example, if we list the nodes $n_p(1) - 1 \rightarrow 0$ for the neutron sector, in the first pass, node 0 via *gsendx* sends ψ' to nodes $n_p(1)-1, \ldots, 2, 1$ in this order. The next node to send its local state ψ' is node 1. Node 1 must wait to receive the vector from node 0 before proceeding to orthogonalize its local ψ' with respect to the wave function vector sent by node 0 and then to broadcast the local ψ'. Therefore, in this scenario, node 1 must wait for all of the destination nodes to receive the vector from node 0, before it can proceed. This represents a bottleneck in the communication sequence. A significant improvement in performance is made when the order of the destination nodes in *ianode* is reversed to increasing order. In this case node 0 broadcasts to nodes $1, 2, \ldots, n_p(1) - 1$ in this order. Since node 1 is the first node to receive ψ' from node 0, it can immediately proceed to process and broadcast its own local ψ', even before all of the destination nodes have finished receiving the vector sent by node 0. For cases with a large number of states, several nodes can actually be broadcasting their local vectors simultaneously. The entire procedure remains naturally sequential and orderly, although many of the operations are being performed simultaneously. These modifications involved filling the array *ianode* with the following:

$$ianode(i - nnode0 + 1, \text{neutron or proton}) = i - 1 , \qquad (27)$$

where $i = nnode0, nnode$,

$$nnode0 = \begin{cases} 1, & \text{neutron,} \\ n_p(1)+1, & \text{proton,} \end{cases}$$

and (28)

$$nnode = \begin{cases} n_p(1), & \text{neutron,} \\ n_p(1)+n_p(2), & \text{proton.} \end{cases}$$

This sequence of communications was deciphered and verified with the help of the Intel performance monitoring tool *paragraph*, which was found to be very useful.

It was found that the use of the Paragon routine *gsendx* is not completely reliable. For example, for more than 100 nodes, the code would sometimes stop and for more than 200 nodes it is difficult to get *gsendx* to work consistently. This is probably due to confusion arising from the simultaneous passing of numerous large messages from various sending nodes to various receiving nodes. Therefore handshaking is introduced for each send and receive that would normally be performed in *gsendx* so as to improve the efficiency of this process. A revised version of *gsendx* was written, which incorporates handshaking between the sending node and the receiving nodes in the following fashion. For the sending routine, the procedure is as follows. After calling *gtsend*, with a positive value of *nnode*, *gtsend* loops through the integer array *ianode*, where, first, a zero length message is sent to the destination node, second, a zero length reply is received indicating the destination node is ready to receive, and then the long message is sent asynchronously. The routine then waits for the last message to be completed before exiting *gtsend*. Hence in this procedure a zero length send and receive are performed in a handshaking manner before the real message is passed. All of the nonsending nodes call *gtsend* to make sure the message labels *msgtype* are kept consistent. In this case *gtsend* is called using a negative value of *nnode*, where *gtsend* again loops through the integer array *ianode*. When the local node *me* corresponds to an entry in *ianode*, the following occurs. First, a zero length message is received from the sending node, second, an asynchronous receive for the long message is posted in preparation, and finally a zero length message is sent to the sending node to indicate the receiving node is ready. A call to *msgwait* is made at the end, so that *gtsend* waits until the long message is completely received.

The handshaking in *gtsend* makes the code much more reliable than when the routine *gsendx* is used. In this form there is no problem executing with up to 250 nodes. Also the use of asynchronous sends and receives makes the net Gram-Schmidt routine much more efficient. The reason is that the receiving nodes post a receive before the message is sent, so that the lengthy message is received directly into the node memory, bypassing the message buffer. Also, the global sending process is performed asynchronously as well and the sending node need not wait for the previous send to be completed before proceeding onto the next send. Results of the increase in efficiency will be shown in Sec. IV.

E. Broadcasts and global summations

Here we discuss some of the algorithms used in performing the communication tasks mentioned above. We have already discussed the communication tasks involved in the Gram-Schmidt process. In addition to Gram-Schmidt procedures we have to perform global sums for the densities and currents given by Eq. (7). In practice, we treat global sums on the iPSC/860 supercomputer differently from the Paragons due to their architectural difference (hypercube versus 2D mesh). For the iPSC/860 supercomputer the basic algorithm is the broadcast algorithm which ensures that messages are transmitted along routes which do not interfere with others and the communication load is distributed in a balanced way. For the hypercube architecture the neighboring nodes are identified by their Gray code [18, 19], which is a binary interconnection scheme where the processors are numbered as decimal numbers, beginning with 0, and arranged such that their binary representations only differ by a single bit location. To perform broadcast and global sums we have used the *subcube broadcast* algorithm [20]. This algorithm has the communication tree as shown in Fig. 1 for an eight node cube. Notice that along each branch binary representations differ only by one bit location. Below is the algorithm we have used on the iPSC/860 supercomputer:

$$\text{send message to nodes } me + k, \quad \forall\, me < k,$$
$$\text{receive message, if } k < me < 2k\ ,$$

$$\text{perform above operation } \forall\, k = 1, 2, 4, 8, \ldots, n$$

where n is the dimension of the cube and *me* denotes the node number. The last argument of the *send* routine is the destination node. On the Paragon supercomputer the broadcast is done by using a global option of the synchronous *csend* routine, which then sends a message to all other nodes. The receiving nodes receive the message in a normal fashion.

The generalization of the above broadcast algorithm can be used to perform global summations on the iPSC/860 supercomputer. This is done by first performing a reverse broadcast by starting from the bottom of the broadcast tree and accumulating the results at node 0. Subsequently, node 0 performs a forward broadcast to distribute the result to all nodes. On the Paragon we again use the resident routine *gdsum* for global double precision summation.

IV. NODAL OPTIMIZATION

The previous section contains discussions about increasing the efficiency of the parallel implementation, especially in regard to decreasing the communication overhead. Effort was also directed in this work to increase the efficiency of the nodal operation, in particular, providing increased vectorization of the code. Typically this

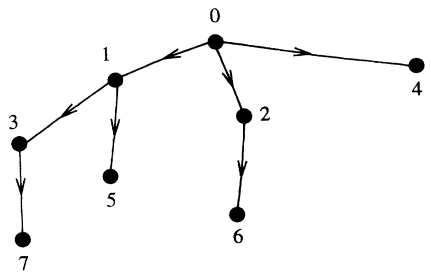

FIG. 1. Broadcast tree for an eight-node cube using the subcube algorithm.

involved disentangling loops resulting in larger vectorizing inner loop structures. An example of the approach that we have taken is presented in this section. In particular we consider the operation of a matrix operator in the y coordinate onto a wave vector. A specific example of such a matrix operator is the gradient operator in the y direction as represented in the collocation spline basis as given in Eq. (17). Such an operation can be written in the following fashion:

$$\psi_{\rm out}(i_x,i_y,i_z,s,k)=\sum_{j_y}Y(i_y,j_y)\psi_{\rm in}(i_x,j_y,i_z,s,k)\ ,$$

$$\forall\ s,k=1,2;\quad i_x\le n_x\ ;\ i_z\le n_z\ .$$

Here the indices s and k correspond to nucleon spin and to real and imaginary parts, respectively. To obtain a larger inner loop structure, a call is made to a subroutine, which reorders the indices in the following fashion:

$$\psi''_{\rm in}(i,i_y)=\psi'_{\rm in}(i_x,i_z,s,k,i_y)=\psi_{\rm in}(i_x,j_y,i_z,s,k)\ ,$$
$$\psi''_{\rm out}(i,i_y)=\psi'_{\rm out}(i_x,i_z,s,k,i_y)=\psi_{\rm out}(i_x,j_y,i_z,s,k)\ .$$

Here the index $i=s\times k\times i_x\times i_z$ and $i\le n$, where $n=4n_xn_z$. With this reordering the matrix operation now becomes

$$\psi''_{\rm out}(i,i_y)=\sum_{j_y}Y(i_y,j_y)\psi''_{in}(i,j_y),\ \forall\ i\le n\ .$$

The inner loop over i is now much larger and so the vectorization becomes much more efficient. In the end the wave vector indices must be reordered back to the original situation. These four operations are performed using efficient subroutine calls.

For collocation grid sizes of 16^3, 20^3, and 24^3, we find that for this particular case the time required for execution is reduced by about 42–43%. For the z direction the improvement is about 48–49% and for the x direction we find about a 35%, 23%, and 14% improvement for the 16^3, 20^3, and 24^3 lattice grid size calculations, respectively. Note that because of the order of the labeling, the corresponding auxiliary routines for the x and z directions are not the same as in the y direction.

V. TIMING STUDIES

For timing comparisons executions on several platforms were performed. On the parallel machines the maximum number of nodes possible for each case was used, where one nucleon state was placed on each node (A is the number of nucleons, equal to the number of nodes), unless otherwise stated. Nuclei with equal number of protons, Z, and neutrons, N, were calculated, where in this situation the calculation and communication time for the proton and neutron sectors will be essentially equivalent. For the vast majority of nuclei N and Z are not equal, and hence the computational burdens of the two sectors will be unequal, where basically the time difference will correspond to the different amount of time spent within the Gram-Schmidt procedure.

Static Hartree-Fock calculations were performed for ^{16}O, ^{32}S, ^{64}Ge, and ^{128}Gd nuclei, where $N=Z$ in these cases. The size of the basis-spline collocation lattice was also varied, where we studied grids of 16^3, 20^3, 24^3, and 26^3 lattices. The code was run for 100 iterations and then continued for an additional 100 iterations. The reason for this procedure was to take into consideration the initial startup time for initializing a calculation as well as the time required for storing and retrieving the wave functions. Also, in the beginning of some calculations it may be necessary to perform many additional iterations of the Poisson solver due to the initial density configuration. It turns out for the cases considered here that these additional times were negligible. Only the times for the initial 100 iterations are shown and discussed.

In Fig. 2, the total CPU time per node is shown for the calculation of ^{16}O as a function of the lattice grid size. Due to memory limitations we were unable to perform calculations with grids larger than 22^3 on the iPSC/860 supercomputer. Although the Paragon supercomputer is a virtual machine, if there is any significant swapping of memory, then the performance on the Paragon deteriorates dramatically. The Paragon models, XP/S 5 and XP/S 35, at ORNL presently have 16 Mbytes/node memory, with about 10 Mbytes available for computational use. The XP/S 35 supercomputer will be expanded in the near future to 32 Mbytes/node. Present plans are for ORNL to obtain a machine with 64 Mbytes/node, which will eventually be expanded to 128 Mbytes/node. The increased memory will be very useful for our purposes, because larger lattices will be required for large exotic nuclei. With present memory limitations on the Paragon supercomputers we are essentially constrained to a maximum of 24^3 lattice sizes.

The total CPU time/node as a function of A is shown in Fig. 3 for fixed lattice sizes. The CPU execution time is retrieved for each node. The "total CPU time/node" as defined in this paper is not the average execution time per node, but the nodal execution time which is the longest. This would then correspond to the amount of wall clock time required for execution with no time sharing. The average nodal execution time and the maximum nodal execution time are in general very close due to necessary

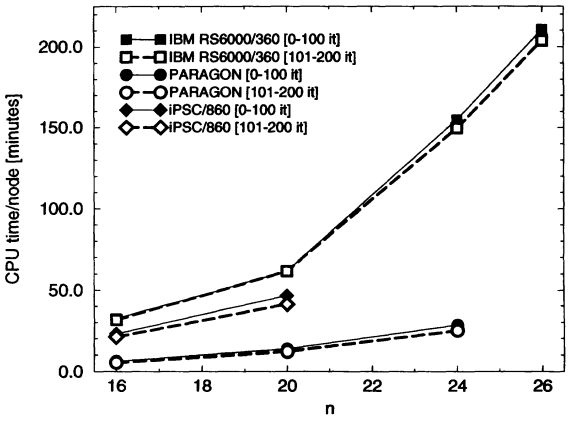

FIG. 2. A timing comparison for calculations of ^{16}O is plotted for different platforms as a function of n, the lattice grid size. The abbreviation *it* denotes iterations.

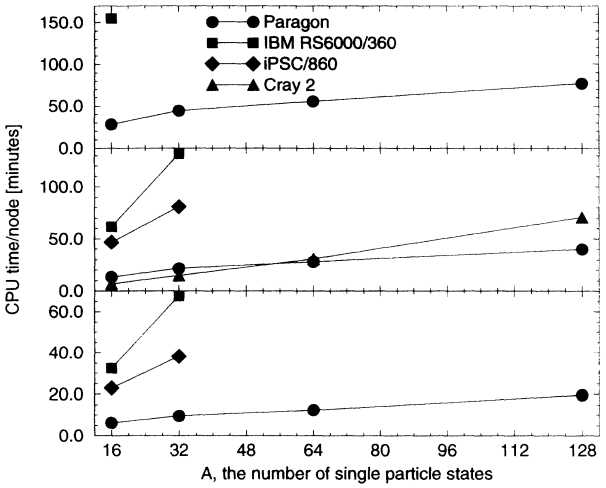

FIG. 3. The total CPU execution time for 100 iterations per node on several platforms is shown as a function of A, the number of single-particle states. For the Paragon and the iPSC/860 parallel computers A is equal to the number of nodes used in these computations. The upper, middle, and lower panels correspond to collocation lattice grid sizes of 24^3, 20^3, and 16^3 points, respectively.

global synchronizing operations. In Fig. 3 the timing results are shown for the Paragon, iPSC/860, and Cray 2 supercomputers and for an IBM RS6000/360 workstation, which has been rated at 22.5 Mflops for double precision Fortran Linpack. Typical calculations involve about 500 iterations and so it is clear that, for large nuclei, sequential machines would become very unwieldy. One can see that the Paragon supercomputer provides a superior platform in comparison to both the IBM workstation and the iPSC/860 supercomputer. For 64 and 128 nodes, the Paragon is faster than the Cray 2 supercomputer. The execution time for the sequential Cray supercomputer is basically linear with A. With expected improvements in the nodal processor speed, the performance of the Paragon should improve and become much faster than the Cray supercomputer.

For large A, the execution time/node appears to increase linearly for $A > 32$, although there seem to be some fluctuations in the calculated CPU time, probably mostly due to changes in the amount of communication traffic on the machine. For example, for $A = 128$ and a 20^3 lattice we obtained a CPU time/node of 38.48 min in one run and 40.23 min in another.

The time used for communication resides basically in two places. The global double precision sums described in Sec. III E take about 10% of the total nodal CPU execution time. This 10% overhead remains consistent when varying the size of the lattice and the number of nodes or wave functions, and even in comparison between the iPSC/860 supercomputer and the Paragon supercomputers. For cases where the number of nodes is $A/2$, then the global sums take about 6–7 % of the total nodal execution time.

The largest amount of communication time is used during the Gram-Schmidt orthogonalization procedure. This procedure cannot be executed in parallel and involves the passing of large messages between the nodes. Since this procedure involves both computation and communication which is in general done sequentially, the timing of the whole Gram-Schmidt procedure will be considered. It is difficult to separate out the communication time, since some nodes only send messages, while other nodes will only receive messages, and most nodes will do a combination of both. Because of the sequential nature of the Gram-Schmidt procedure, for both the neutron and proton sectors, before the last node in each sector can process its local wave function it must wait for all of the other nodes in the sector to first complete orthogonalizing their local wave vectors and to pass this wave vector to the last node. Hence the last node will have a great deal of idle time, while the first node will complete its Gram-Schmidt procedure quickly and take much less time than the last node. For $N \neq Z$ nuclei the sectors with the most nucleons may take much more time than the other sectors. In some cases this difference can be rather large.

In Fig. 4 the maximum and minimum nodal time spent within the Gram-Schmidt routine is shown to illustrate the inherently sequential nature of this procedure. Times are also shown for the iPSC/860 and Cray 2 supercomputers and the sequential IBM workstation. As the number of nodes, A, is increased, the maximum and minimum Gram-Schmidt execution time increases linearly. This scaling feature will be discussed in more detail later in this section. It is interesting to note that the Gram-Schmidt operation time spent for the Cray 2 supercomputer with $A = 128$ is greater than the corresponding Paragon time. The fraction of the total nodal execution time is displayed in Fig. 5. These fractions are obtained by dividing the times given in Fig. 4 by the corresponding nodal execution times given in Fig. 3. As A increases, it is clear that the Paragon supercomputer

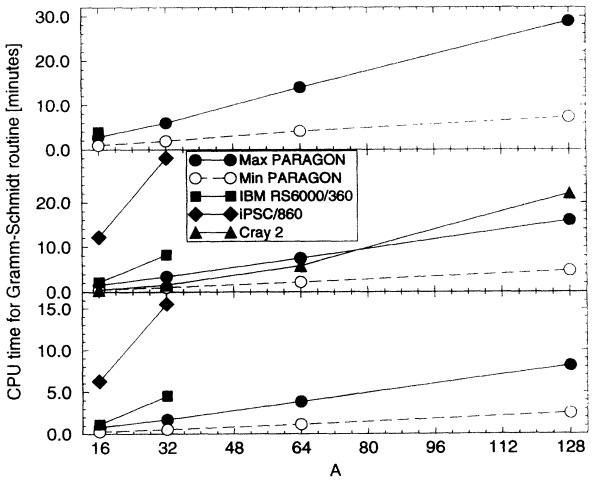

FIG. 4. The execution CPU time per node for 100 iterations in minutes for the Gram-Schmidt routine is plotted. The upper, middle, and lower panels correspond to collocation lattice grid sizes of 24^3, 20^3, and 16^3 points, respectively.

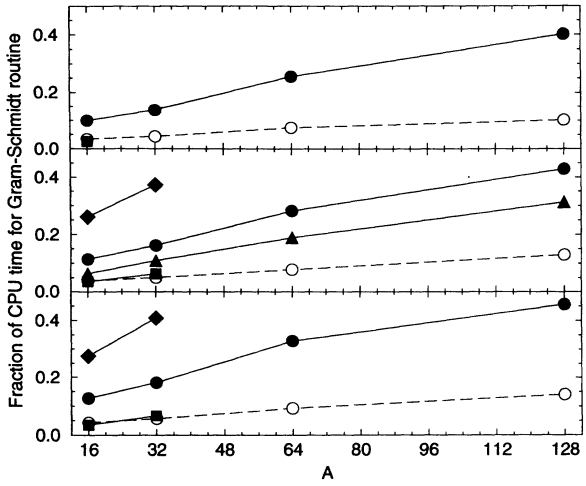

FIG. 5. The fraction of time spent in the Gram-Schmidt routine during execution of the code is shown. The curve labels are the same as those given in Fig. 4. The upper, middle, and lower panels correspond to collocation lattice grid sizes of 24^3, 20^3, and 16^3 points, respectively.

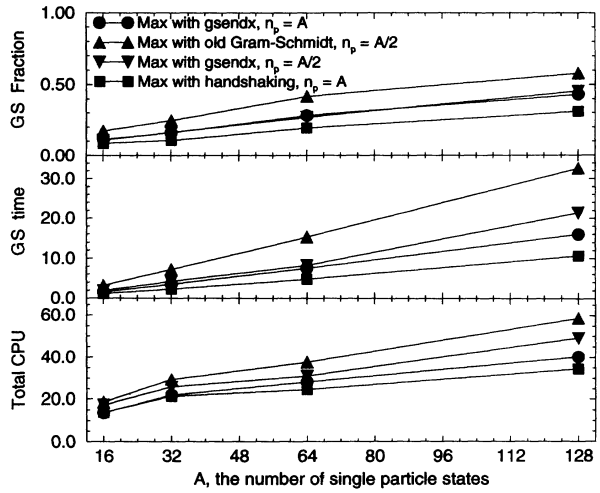

FIG. 6. A comparison of the various modifications to the Gram-Schmidt (GS) routine as a function of A, the number of nucleons, is shown. The collocation lattice grid is fixed to 20^3 points. The results shown are the maximum nodal times for execution on the Paragon supercomputer. The curve with the squares corresponds to the calculation which uses the modified *gsendx* with handshaking. The bottom panel shows the total execution time/node in minutes, the middle panel shows the nodal time spent within the GS routine, and the top panel shows the fraction of time spent in the GS routine in the same fashion as in Fig. 5.

is much more efficient in communication in comparison with the iPSC/860. On the Paragon supercomputer for large A the maximum fraction of time spent in the Gram-Schmidt procedure is about 40–50 %, thus creating a significant overhead for the program.

The modifications to the Gram-Schmidt routine as described earlier provide considerable improvement in execution time and efficiency. The timing results shown in Figs. 2–5 use the improved version of the Gram-Schmidt procedure with the Paragon routine *gsendx*. The old Gram-Schmidt routine, which uses a general broadcast to all other nodes, is only able to use a maximum of $\min(Z, N)$ nodes, while the modified Gram-Schmidt procedure is able to use $A = N + Z$ nodes. For $Z = N = A/2$ nodes timing comparisons using the old and new Gram-Schmidt routines are shown in Fig. 6. Also shown are results with the new Gram-Schmidt routine using A nodes and the case where *gtsend* is used in place of *gsendx*. A 20^3 lattice grid is used in these computations. Recall that the new Gram-Schmidt routine maintains separate proton and neutron sectors with no communication between the two sectors. The old Gram-Schmidt procedure lays the neutron sector on top of the proton sector where both sectors span all of the nodes. In Fig. 6 the triangles pointing up correspond to the maximum nodal times for the old Gram-Schmidt routine. The triangles pointing down use the new Gram-Schmidt routine with the solid symbols representing the maximum nodal times. In the middle panel the modified Gram-Schmidt procedure reduces the Gram-Schmidt execution by about 50% and hence reduces the total nodal CPU execution time. This savings becomes especially important for very large nuclei.

In comparison with the A node calculation represented by the circles, the minimum nodal time spent in the new Gram-Schmidt routine is the same as the $A/2$ node calculation using the new Gram-Schmidt routine. To make a comparison with the maximum nodal time in the Gram-Schmidt routine we need to consider the amount of communication and computation involved in the time the last node in each sector must spend in the Gram-Schmidt routine. The amount of computation that the last node must either wait for or perform is essentially the same in these two cases, so let us consider the difference in communication time. Given N neutrons, for the A node case the last node must perform or wait for $N-1$ sends and receives. For the A/2 node case the last node is involved with $2\left(\frac{N}{2}-1\right) = N-2$ sends and receives. So, even though it would seem that with fewer nodes and a sequential routine, there should be less communication, this is not the case. The actual communication time involved with the last node is essentially the same for these two cases. This is reflected in Fig. 6, except for the $A = 128$ point, which is probably high due to fluctuations in traffic on the machine. Since the amount of communication involving the last node is proportional to N, the time spent within the Gram-Schmidt routine should scale linearly, which is precisely the pattern observed in Fig. 4.

Although the Gram-Schmidt procedure in a strictly sequential sense does not increase linearly, but geometrically, the behavior seen in Fig. 6 can be understood with the following discussion. As described in Sec. III D the modified new Gram-Schmidt routine uses *gsendx* in a pipelined fashion. By having a node broadcast the local wave vector in a particular order, i.e., to the next node in

the orthogonalizing sequence, one can make the communication more efficient and reduce the idle time. For example the following sequence for the new Gram-Schmidt routine can be stated as follows. Node 0 computes, then broadcasts to nodes $1,2,\ldots,N-1$. Since node 1 is the next node in the sequence and is the first to receive the wave vector from node 0, node 1 can immediately receive, compute, and then broadcast its local wave vector to nodes $2,3,\ldots,N-1$. Node 2 then proceeds as well, etc. Hence the broadcasting is performed in a pipeline fashion, since for large N, there can be several wave vectors being broadcast from several nodes at the same time, all of which will eventually be received by node $N-1$. Also, while, for example, node 1 is receiving and computing, the other nodes are also receiving and computing at essentially the same time, with some delay. Thus the computations within the Gram-Schmidt routine are also performed in a pipeline fashion and hence the computation time should increase linearly with A as well as the communication time. In the top panel of Fig. 6 it can be seen that the percentage of nodal time spent in the new Gram-Schmidt routine is significantly reduced in comparison with the old Gram-Schmidt results. For larger nuclei, for example, $A = 238$, $Z = 92$ uranium, this savings becomes important.

The use of *gtsend* in place of *gsendx* provides additional improvement in performance. Not only does the code become much more reliable, but the actual execution is much more efficient. In the middle panel, one can see that the maximum nodal time for the Gram-Schmidt procedure is reduced by about one third. This reduces the fraction of time spent within the Gram-Schmidt routine from about 43% to about 30%. In summary, all of the modifications performed on the Gram-Schmidt routine provide an improvement of about a factor of 3 and a net savings of about 50% in total nodal execution time. The resulting improved performance is comparable to or better than that found on the sequential Cray 2 supercomputer.

VI. CONCLUSIONS

Massively parallel platforms, such as the Paragon and the iPSC/860 supercomputers, provide a much improved vehicle for performing mean-field calculations. Because of the greater computational capabilities, calculations of large complex and exotic nuclear many-body systems can now proceed with a greater degree of sophistication than has previously been possible. A program to perform static Hartree-Fock mean-field calculations using a full three-dimensional basis-spline collocation lattice has been developed with no spatial or time-reversal symmetries imposed. This program has been ported to the Paragon and the iPSC/860 supercomputers. In addition, an algorithm has been developed which takes advantage of some of the features of the Paragon supercomputer to streamline the communication intensive Gram-Schmidt orthogonalization routine by pipelining the message passing and some of the computations.

ACKNOWLEDGMENTS

This research has been supported in part by the U.S. Department of Energy (DOE) Office of Scientific Computing under the High Performance Computing and Communications Program (HPCC), as a Grand Challenge titled the Quantum Structure of Matter, and in part by DOE under Contract No. DE-AC05-84OR21400 managed by Martin Marietta Energy Systems, Inc., and under Contract No. DE-FG05-87ER40376 with Vanderbilt University. Some of the numerical calculations were carried out on the Intel Paragon and iPSC/860 parallel computers at the Oak Ridge National Laboratory.

[1] K. T. R. Davies, K. R. S. Devi, S. E. Koonin, and M. R. Strayer, in *Treatise on Heavy Ion Science*, edited by D. A. Bromley (Plenum, New York, 1985), Vol. 3, p. 3.

[2] A. S. Umar and M. R. Strayer, Comput. Phys. Commun. **63**, 179 (1991).

[3] C. Bottcher, G. J. Bottrell, and M. R. Strayer, Comput. Phys. Commun. **63**, 63 (1991).

[4] A. S. Umar, M. R. Strayer, R. Y. Cusson, P.-G. Reinhard, and D. A. Bromley, Phys. Rev. C **32**, 172 (1985).

[5] D. H. Tiszauer and K. C. Kulander, Comput. Phys. Commun. **63**, 351 (1991).

[6] B. Jackson, Comput. Phys. Commun. **63**, 154 (1991).

[7] A. S. Umar, M. R. Strayer, P.-G. Reinhard, K. T. R. Davies, and S.-J. Lee, Phys. Rev. C **40**, 706 (1989).

[8] K. T. R. Davies and S. E. Koonin, Phys. Rev. C **23**, 2042 (1981).

[9] P. Hoodbhoy and J. W. Negele, Nucl. Phys. **A288**, 23 (1977).

[10] S. E. Koonin, K. T. R. Davies, V. Maruhn-Rezwani, H. Feldmeier, S. J. Krieger, and J. W. Negele, Phys. Rev. C **15**, 1359 (1977).

[11] A. S. Umar, J. Wu, M. R. Strayer, and C. Bottcher, J. Comput. Phys. **93**, 426 (1991); C. Bottcher and M. R. Strayer, Ann. Phys. (N.Y.) **175**, 64 (1987).

[12] A. S. Umar, M. R. Strayer, J.-S. Wu, D. J. Dean, and M. C. Güçlü, Phys. Rev. C **44**, 2512 (1991).

[13] K. J. Schafer, N. H. Kwang, and J. D. Garcia, Comput. Phys. Commun. **63**, 306 (1991).

[14] J. C. Wells, V. E. Oberacker, A. S. Umar, C. Bottcher, M. R. Strayer, J.-S. Wu, and G. Plunien, Phys. Rev. A **45**, 6296 (1992).

[15] D. J. Dean, C. Bottcher, and M. R. Strayer, Int. J. Mod. Phys. (to be published).

[16] C. De Boor, *Practical Guide to Splines* (Springer-Verlag, New York, 1978).

[17] C. Bottcher, M. R. Strayer, A. S. Umar, and P.-G. Reinhard, Phys. Rev. A **40**, 4182 (1989).

[18] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors* (Prentice-Hall, Englewood Cliffs, N.J., 1988), Vol. I, p. 261.

[19] T. F. Chan and Y. Saad, *IEEE Trans. Comput.* **C-35**, 969 (1986).

[20] G. C. Fox, S. W. Otto, and A. J. G. Hey, *Parallel Computing* **4**, 17 (1987).