Available online at www.sciencedirect.com

ScienceDirect

Journal of Computational Physics 219 (2006) 185–197

JOURNAL OF COMPUTATIONAL PHYSICS

www.elsevier.com/locate/jcp

Parallel implementation of efficient preconditioned linear solver for grid-based applications in chemical physics. I: Block Jacobi diagonalization

Wenwu Chen, Bill Poirier [*]

Department of Chemistry and Biochemistry, and Department of Physics, Texas Tech University, Box 41061, Lubbock, TX 79409-1061, USA

Received 6 September 2005; received in revised form 31 March 2006; accepted 24 April 2006
Available online 23 June 2006

Abstract

Linear systems in chemical physics often involve matrices with a certain sparse block structure. These can often be solved very effectively using iterative methods (sequence of matrix–vector products) in conjunction with a block Jacobi preconditioner [Numer. Linear Algebra Appl. 7 (2000) 715]. In a two-part series, we present an efficient parallel implementation, incorporating several additional refinements. The present study (paper I) emphasizes construction of the block Jacobi preconditioner matrices. This is achieved in a preprocessing step, performed prior to the subsequent iterative linear solve step, considered in a companion paper (paper II). Results indicate that the block Jacobi routines scale remarkably well on parallel computing platforms, and should remain effective over tens of thousands of nodes.
© 2006 Elsevier Inc. All rights reserved.

PACS: 33.20.−t; 82.20.−w

Keywords: Sparse matrix; Preconditioning; Block Jacobi; Chemical physics; Parallel computing; Eigensolver; Linear solver

1. Introduction

The symmetric eigenvalue/eigenvector problem, $(\mathbf{H} - \lambda\mathbf{I})\mathbf{x} = 0$, and linear solve problem, $\mathbf{w} = (\mathbf{H} - \lambda\mathbf{I})^{-1}\mathbf{v}$ (where $\mathbf{H}$ and $\mathbf{I}$ are $N \times N$ matrices, $\mathbf{x}$, $\mathbf{w}$, and $\mathbf{v}$ are vectors, and λ is a scalar), recur countless times in scientific and engineering disciplines. In molecular and chemical physics, these operations can be directly associated with the computation of essentially all dynamical quantities of interest – vibrational and rovibrational energy levels and wavefunctions, resonance energies and lifetimes [1,2], scattering cross-sections, cumulative reaction probabilities [3–6], and chemical reaction rates. Typically, the matrix $\mathbf{H}$ is the representation of the Hamiltonian differential operator, $\hat{H}$, in some finite basis of N orthonormal functions, presumed to span the relevant portion of Hilbert space. However, one can also choose as representational "basis" a discrete set of N grid

DOI of original article: 10.1016/j.jcp.2006.03.031.
[*] Corresponding author. Tel.: +1 806 742 3099; fax: +1 806 742 1289.
E-mail addresses: wenwu1012@yahoo.com (W. Chen), Bill.Poirier@ttu.edu (B. Poirier).

0021-9991/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2006.04.012

points distributed over position space, as is routinely done in the engineering and applied mathematics fields to simulate partial differential equations (PDEs). Despite offering many advantages, even discrete grid representations can lead to inordinate computational effort, especially for chemical physics applications. Accordingly, the goal of this paper (paper I) and its companion (paper II [7]) is to develop efficient parallelization strategies.

The precise determination of the grid-based matrix **H** requires specification of the particular discretization scheme employed, e.g. finite difference [8], discrete variable representation (DVR) [9–15], optimized DVR's [16–22], etc. Our particular interest is the quantum dynamics of molecular systems, for which the appropriate PDE to be solved is the nuclear motion Schrödinger equation, and DVR discretization is most commonly employed. However, as described in detail in Appendix A, the particular choice of discretization is largely immaterial, as *all* such choices present similar advantages and challenges for the numerical methods described here.

The primary advantage of a discretized representation is *sparsity*, meaning that the majority of the matrix elements of **H** are zero. A second advantage applies if the grid points are laid out in a rectilinear (but not necessarily uniform) lattice, as in Fig. 1. In this case, the resultant **H** matrix *must* adopt the highly structured block form of Fig. 2 (Appendix A, Ref. [23]), which we term the "**A** matrix form". Sparsity opens the door to a host of standard numerical techniques, none of which, unfortunately, tends to be very effective for the applications described above. In particular, the maximum distance between any two points in the graph [8] of a generic **A** matrix is two, thus invalidating the nested dissection method. The profile of an **A** matrix (Refs. [7,24]) is almost as large as N, and cannot be significantly reduced via any permutation of rows and columns – implying that reordering methods such as reverse Cuthill-McKee [25] will not be very effective. Local pivoting methods such as minimum degree are also ineffectual, because **A** matrices present a worst-case scenario with respect to tie-breaking. Sparse iterative methods [26], i.e. those that operate via a sequence of matrix–vector products, tend to be effective at low energies, λ; however, the number of iterations required to achieve numerical convergence, M, becomes prohibitively large when λ is well in the interior of the spectrum of **H** [27]. This pathological behavior is due to the spectral density of the **H** matrices involved, which tends to increase very dramatically with energy for molecular Hamiltonians.

Preconditioning is a numerical technique that can reduce M substantially, provided that an approximation **P** to the matrix ($\mathbf{H} - \lambda\mathbf{I}$) can be constructed, such that matrix–vector products with $\mathbf{P}^{-1}$ are computationally inexpensive (further details may be found in paper II). Unfortunately, *none* of the standard preconditioners, including successive over-relaxation (SOR), Gauss–Seidel, and Jacobi [8], are effective for the above applica-

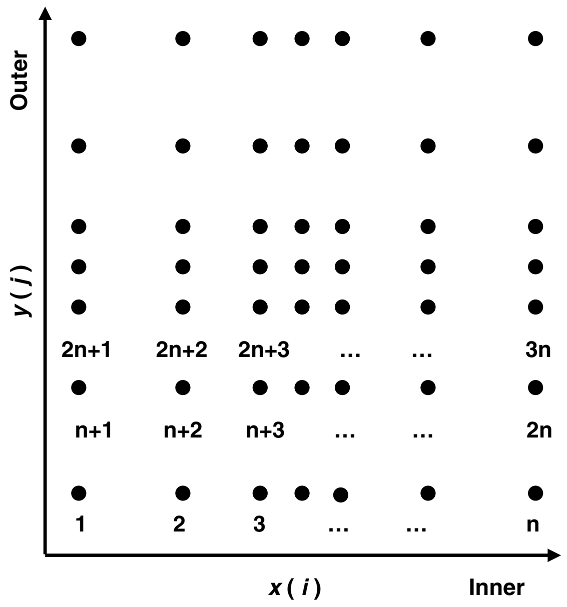


Fig. 1. Schematic of structured rectilinear grids commonly used in PDE applications. Shown here is a two-dimensional case, demonstrating explicitly the role of "inner" x and "outer" y dimensions on (lexicographical) grid point ordering. The two dimensions may be actual spatial dimensions, or "effective" dimensions obtained via dimensional combination.

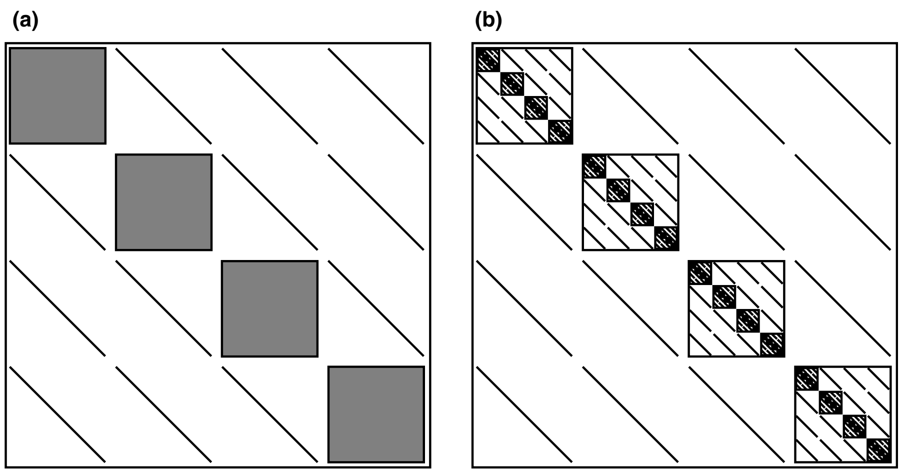


Fig. 2. Sparsity pattern for Hamiltonian matrix representations on multidimensional rectilinear structured grids (Fig. 1): (a) $d = 2$ dimensions; (b) $d > 2$ dimensions, for which diagonal blocks are self-similar to the whole. For $d > 2$, two-tiered block Jacobi diagonalization transforms the original (b) matrix to the less sparse form of (a), thus introducing fill-in. In contrast, recursive block Jacobi diagonalization preserves the form of (b).

tions. Jacobi preconditioning, for instance, for which **P** is simply taken to be the diagonal matrix elements of $(\mathbf{H} - \lambda\mathbf{I})$, requires diagonally-dominant matrices. In molecular Hamiltonians, however, the diagonal contribution (arising primarily from the potential energy) and off-diagonal contribution (the kinetic energy) are roughly comparable. A better preconditioning strategy – the precursor to that adopted here – is to exploit the natural block structure of the **A** matrix form. In particular, taking the diagonal blocks of Fig. 2(a) to be **P**, one obtains the standard block-Jacobi preconditioner. Though somewhat improved, this approach is still not very effective, because the off-block-diagonal elements are quite substantial (Ref. [23] presents a more detailed discussion).

If significant progress is to be made, a substantially different approach must be found. One of the authors (Poirier) has spent over a decade developing effective preconditioners for chemical physics applications. By far, the most efficient at reducing M (usually by orders of magnitude) are the optimal separable basis (OSB) preconditioners [23,28–30]. A detailed description may be found in the above citations, but the basic idea is straightforward: one applies an orthogonal transformation to **H**, known as "block Jacobi diagonalization" (Section 2) such that (1) the **A** matrix form is preserved, and (2) the off-block-diagonal matrix elements are minimized. One then applies block-Jacobi preconditioning in the transformed representation. Construction of the OSB preconditioner matrix **P** (and the related orthogonal transformation matrix **V**) requires nontrivial computational effort, as compared with the standard preconditioner choices described above. However, preconditioner construction need occur only once per calculation, in a preprocessing step performed prior to iterative solution. The preprocessing CPU cost is greater than that of a single matrix–vector product (Section 2), but typically less than the collective cost for all M iterations.

OSB techniques have proven to be extremely effective at reducing M for real molecular applications [23,27–33]. However, they do little to combat the primary numerical difficulty, which is exponential scaling of N with respect to system dimensionality, d. More specifically, $N = n^d$, where n is the number of grid points per dimension, and $d = 3(A - 1)$ where A is the number of atoms. Consequently, fully quantum dynamical treatments have traditionally been limited to small molecules (3–5 atoms), despite much interest in larger systems. With the recent advent of massively parallel terascale and petascale computing clusters, distributing the computational burden among hundreds to tens-of-thousands of CPUs, it is natural to consider parallel implementations of the above methodologies.

In this two-part series, we therefore develop efficient strategies for parallelizing not only the standard iterative solver routines, but also the specialized OSB preconditioner construction routines. Paper II deals with the former, focusing on the fundamental parallel matrix–vector product operation itself. This paper (I) addresses the latter by presenting an efficient parallel implementation of block Jacobi diagonalization. The ultimate goal of this effort is to develop generalized, modular, parallel codes that can be readily applied in a variety of disciplines. This requires that the algorithms must first be generalized for recursive application to arbitrary system dimensionality, d. To assess parallel and system size scalability, a model molecular system has been selected, for which d can be easily modified.

The organization of paper I is as follows. Section 2 overviews the serial implementation of recursive block Jacobi diagonalization. Section 3 describes the parallel implementation, discussing domain decomposition, data distribution, and load balancing. Section 4 presents performance benchmarks for parallel block Jacobi diagonalization applied to the model system described above. Speedup and parallel efficiency data are provided and analyzed for a variety of data configurations. To improve performance of the basic matrix–vector product operation (paper II), the dimensional combination technique is introduced in Section 4.3, wherein the effects on parallel efficiency and total compute time are also discussed. A joint summary and concluding discussion for both papers will be provided in paper II.

2. Block Jacobi diagonalization

2.1. Two-tiered version

The heart of OSB preconditioning is block Jacobi diagonalization [23,27]. This is most straightforward to apply when **H** is an **A** matrix, i.e.: (1) is partitioned into square blocks of identical size; (2) has diagonal off-diagonal blocks; (3) has identical diagonal blocks, apart from the diagonal matrix elements. As discussed in Section 1 and Appendix A, **A** matrices are very common, and necessarily result when rectilinear structured grids are used, or even unstructured grids if dimensional combination (Section 4.3) is employed [28–30].

For the $d = 2$ case (Appendix A), **H** corresponds to Fig. 2(a). In order to preserve the **A** matrix form, the block Jacobi orthogonal transformation **V** must satisfy

$$\mathbf{V} = \mathbf{I}_x \otimes \mathbf{v}_y, \tag{1}$$

where $\mathbf{v}_y$ is itself orthogonal. In particular, $\mathbf{v}_y$ is chosen so as to minimize the off-block-diagonal contribution of the transformed **H** matrix. Thus,

$$\mathbf{V}^{\mathrm{T}}\mathbf{H}\mathbf{V} = \mathbf{H}^{\mathrm{D}} + \mathbf{H}^{\mathrm{O}}, \tag{2}$$

where $\mathbf{H}^{\mathrm{D}}$ and $\mathbf{H}^{\mathrm{O}}$ comprise the (transformed) diagonal and off-diagonal blocks, respectively, and $\|\mathbf{H}^{\mathrm{O}}\|$ is minimized over $\mathbf{v}_y$. The resultant preconditioner, $\mathbf{H}^{\mathrm{D}}$, is in effect an optimized adiabatic approximation [32]. A detailed mathematical and algorithmic description may be found in the citations [23,28–30].

The above two-tiered block Jacobi procedure can also be applied when $d > 2$ [30,32,33], provided dimensional combination is employed. In particular, the d dimensions are partitioned into inner and outer subsets, treated, respectively, as the lexicographically combined 'x' and 'y' dimensions, using the notation of Appendix A. Thus,

$$i = (i_1, i_2, \ldots, i_l) \quad \text{and} \quad j = (i_{l+1}, \ldots, i_d). \tag{3}$$

However, much of the original sparsity is lost under Eq. (2), which in effect transforms the original Fig. 2(b) matrix into a Fig. 2(a) matrix. If $l = d/2$ for example, the transformed matrix has $\mathcal{O}(n^{3d/2})$ rather than $\mathcal{O}(n^{d+1})$ non-zero matrix elements.

2.2. Recursive version

An alternate, recursive version of block Jacobi diagonalization can be formulated that preserves the original $\mathcal{O}(n^{d+1})$ sparsity pattern. This is achieved by first applying two-tiered block Jacobi to the outermost dimen-

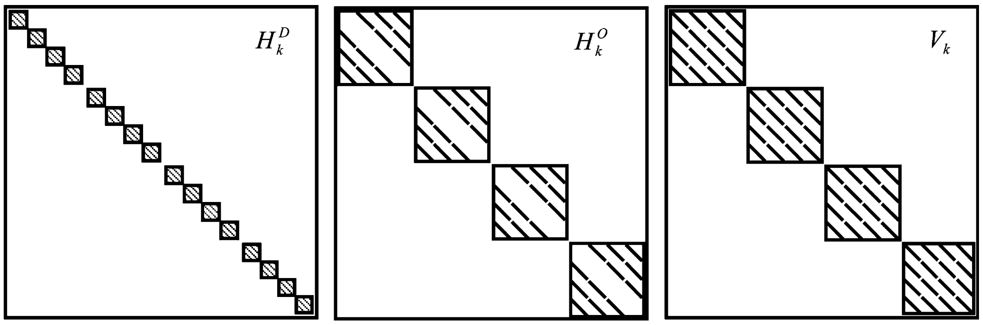


Fig. 3. Sparsity patterns of the diagonal block ($\mathbf{H}_k^{\mathrm{D}}$) and off-diagonal block ($\mathbf{H}_k^{\mathrm{O}}$) contributions to the Hamiltonian matrix, as obtained from applying block Jacobi diagonalization at some intermediate dimension k. The k-level transformation matrix, $\mathbf{V}_k$, is also indicated. On average, the non-zero elements of $\mathbf{H}_k^{\mathrm{O}}$ are much smaller in magnitude than those of $\mathbf{H}_k^{\mathrm{D}}$. Only the diagonal elements of $\mathbf{H}_k^{\mathrm{D}}$ differ from (small) block to block, whereas the elements of $\mathbf{H}_k^{\mathrm{O}}$ are all different. For $\mathbf{V}_k$, there is variation across blocks, but all matrix elements within a (small) block are identical.

sion (i.e. $l = d - 1$), and then recursively to each of the resultant diagonal blocks. The first and outermost transformation $\mathbf{V}_d$ takes the diagonal-block form:

$$\mathbf{V}_d = \mathbf{I}_1 \otimes \mathbf{I}_2 \cdots \mathbf{I}_{d-1} \otimes \mathbf{v}_d. \tag{4}$$

The next, $(d-1)$-level transformation is block-diagonal:

$$\mathbf{V}_{d-1} = \mathbf{I}_1 \otimes \mathbf{I}_2 \cdots \mathbf{I}_{d-2} \otimes (\mathbf{V}_{d-1}^{i_d=1} \oplus \cdots \oplus \mathbf{V}_{d-1}^{i_d=n}). \tag{5}$$

The remaining transformations, $\mathbf{V}_k$, are generated in like manner, down to the innermost dimension, $k = 1$.

The following is a numerical recipe for implementing the recursive block Jacobi scheme:

$$\begin{aligned}
\mathbf{V}_d^{\mathrm{T}}\mathbf{H}\mathbf{V}_d &= \mathbf{H}_d^{\mathrm{D}} + \mathbf{H}_d^{\mathrm{O}} \\
\mathbf{V}_{d-1}^{\mathrm{T}}\mathbf{H}_d^{\mathrm{D}}\mathbf{V}_{d-1} &= \mathbf{H}_{d-1}^{\mathrm{D}} + \mathbf{H}_{d-1}^{\mathrm{O}} \\
&\vdots \\
\mathbf{V}_1^{\mathrm{T}}\mathbf{H}_2^{\mathrm{D}}\mathbf{V}_1 &= \mathbf{H}_1^{\mathrm{D}}.
\end{aligned} \tag{6}$$

In Eq. (6) above, $\mathbf{H}_d^{\mathrm{D}}$ and $\mathbf{H}_d^{\mathrm{O}}$ are the block-diagonal and off-block-diagonal Hamiltonian contributions after the outermost $\mathbf{V}_d$ transformation has been applied. At the next level, *only* the $\mathbf{H}_d^{\mathrm{D}}$ contribution is further transformed, to yield $\mathbf{H}_{d-1}^{\mathrm{D}}$ and $\mathbf{H}_{d-1}^{\mathrm{O}}$, etc. The final $\mathbf{V}_1$ transformation diagonalizes the innermost blocks, resulting in $\mathbf{H}_1^{\mathrm{O}} = 0$. The other $\mathbf{H}_k^{\mathrm{O}}$'s, though not zero, are minimized by the block Jacobi procedure. It can be shown that the storage required for the $\mathbf{H}_k^{\mathrm{D}}$, $\mathbf{H}_k^{\mathrm{O}}$ and $\mathbf{V}_k$ matrices scales as n^{d+1} or less; the corresponding sparsity patterns are indicated in Fig. 3. The total number of CPU operations to implement Eq. (6) is $\mathcal{O}(n^{d+2})$ [23], as compared to $\mathcal{O}(n^{2d})$ for the two-tiered approach with $l = d/2$.

Some of the matrices computed in the preprocessing step above must be stored for the subsequent preconditioned iterative solve step (paper II). In particular, the $\mathbf{V}_k$ are needed to transform between the grid and OSB representations, and $\mathbf{H}_1^{\mathrm{D}}$ is used to compute the preconditioner $\mathbf{P}$. One scheme (OSBW preconditioning [7,27,31,34]) also requires the $\mathbf{H}_k^{\mathrm{O}}$, in order to compute Hamiltonian matrix elements in the OSB representation via

$$\mathbf{H}_{\mathrm{OSB}} = \mathbf{H}_1^{\mathrm{D}} + \mathbf{V}_1^{\mathrm{T}}\mathbf{H}_2^{\mathrm{O}}\mathbf{V}_1 + \cdots + \mathbf{V}_1^{\mathrm{T}}\mathbf{V}_2^{\mathrm{T}} \cdots \mathbf{V}_{d-1}^{\mathrm{T}}\mathbf{H}_d^{\mathrm{O}}\mathbf{V}_{d-1} \cdots \mathbf{V}_2\mathbf{V}_1. \tag{7}$$

3. Parallel implementation

3.1. Domain decomposition

The first step towards a parallel implementation is the determination of a suitable domain decomposition strategy [35]. In many PDE applications, the locality of grid point coupling lends itself well to the standard specification of domains as geographical regions of the grid. This approach would fare extremely poorly for generic **A** matrices, however, which incorporate coupling across the entire spatial extent of the grid (see also Section 1). We are thus motivated to develop our own non-standard domain decomposition scheme.

Given the recursive nature of Eq. (6) and the self-similar block structure of the matrices involved (Figs. 2(b) and 3), it is natural to associate domains with matrix blocks at different levels of the dimension hierarchy, with the largest domains corresponding to the highest dimension index values. Domains are thus geographical regions of the *matrix*, rather than the grid. The matrix elements belonging to each domain are defined as follows:

$$\begin{array}{ll} \text{level } d & \text{all} \\ \text{level } d-1 & i_d = i'_d \\ \text{level } d-2 & i_d = i'_d \text{ and } i_{d-1} = i'_{d-1} \\ \vdots & \\ \text{level } 1 & i_k = i'_k \text{ for all } k > 1. \end{array} \tag{8}$$

The situation is represented in Fig. 4(a). Note that the $\mathbf{H}_k^{\mathrm{O}}$ and $\mathbf{V}_k$ matrices are *block-diagonal* down to level $k+1$, and *diagonal-block* at the lower levels. This is very advantageous from the perspective of minimizing parallel communication. If the total number of nodes is $p = n^{d-s}$, for some integer s with $0 < s < d$, then s represents the k-level value below which all processing occurs on a single node without communication, and no further domain decomposition is needed. The scheme can also incorporate topographical relationships among the p nodes, but we do not consider such a possibility here.

3.2. Data distribution

Distributing data as evenly as possible among the p nodes is a primary consideration in a parallel implementation, and one that is greatly facilitated by the domain decomposition of Section 3.1. For the lower, $k \leqslant s$ levels, whole diagonal blocks of $\mathbf{H}_k^{\mathrm{O}}$ and $\mathbf{V}_k$ (i.e. $i_l = i'_l$ for all $l > k$) are stored on individual processors, in contiguous groups of n^{s-k} blocks each, as indicated in the bottom of Fig. 4(b).

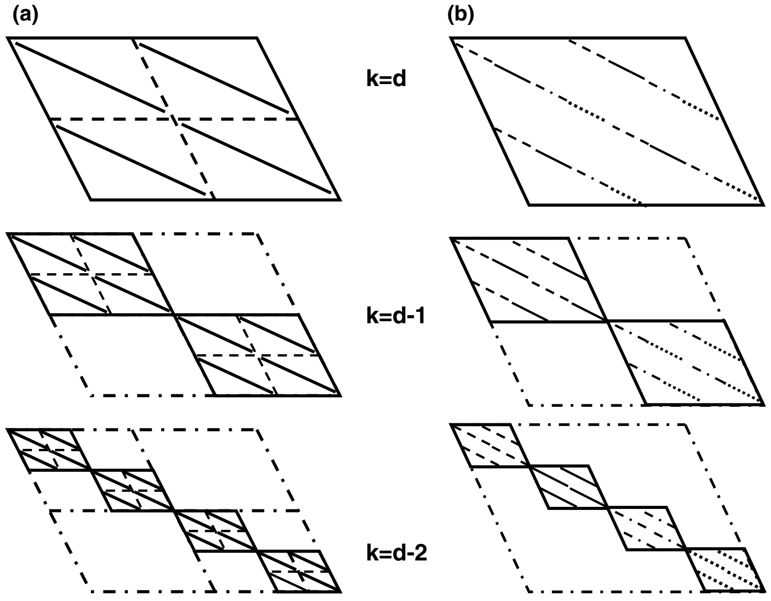


Fig. 4. Schematic of domain decomposition (a) and data distribution (b) as employed by the parallel implementation at the highest three dimensional levels, $k = d$, $k = d-1$, and $k = d-2$ (with $n = 2$ and $s = d-2$). The figure depicts *matrices* rather than grids. Diagonal lines indicate nonzero matrix elements to be stored. At each level in (a), the domains are indicated by solid squares. In (b), the four different diagonal line types (dashed, solid, dot-dashed, and dotted) indicate how data at each level would be distributed across $p = 4$ nodes.

For $k > s$, there are multiple *nodes* per *block* (n^{k-s}), rather than the other way around. A "block-row distribution" scheme is employed [24], for which all matrix elements of a given block row (really a *sub*-block row) are stored on the same node. A block row size of n^{s-1} is used, resulting in a "cyclical", noncontiguous storage of block rows, as indicated in the upper parts of Fig. 4(b). Thus, the first n^{s-1} rows of a given $\mathbf{H}_k^{\mathrm{O}}$ or $\mathbf{V}_k$ block are stored on node 1, the next n^{s-1} rows on node 2, etc., up to node n^{k-s}. At this point, only one part in $n = n^k/(n^{s-1}n^{k-s})$ of the block has been stored, so the next n^{s-1} rows are stored on node 1, starting a second cycle. After n complete cycles, each node stores n block rows of size n^{s-1}.

The great advantage of the above data distribution scheme is that matrix–matrix multiplications with $\mathbf{H}_k^{\mathrm{O}}$ and $\mathbf{V}_k$ at a given level k may be performed without *any* internode communication (this is also true of matrix–vector multiplications, as is exploited in paper II). However, total storage requirements per matrix are $\mathcal{O}(n^{d+1})$, which is wasteful for the $\mathbf{V}_k$ matrices, in the sense that many matrix elements are repeated, and only n^{d-k+2} distinct values must be stored. In practice, therefore, we find it convenient to duplicate all of the $\mathbf{V}_k$ data on all nodes when $k > s$, for which the largest storage required is only $\mathcal{O}(pn)$. Obviously, this still allows for matrix–matrix multiplications without communication.

3.3. Implementation, communication, and load balancing

Implementation of recursive block Jacobi diagonalization consists of two primary operations: (1) performing the matrix–matrix multiplications of Eq. (6); (2) computing the $\mathbf{v}_k$ matrices as used in Eqs. (4) and (5), etc. Since the sparsity structure of $\mathbf{H}_{k+1}^{\mathrm{D}}$ is effectively like that of $\mathbf{V}_k$, it is clear from the discussion in Section 3.2 that (1) can be performed in parallel without any communication. Note that $\mathbf{H}_k^{\mathrm{O}}$ may be immediately discarded once the k-level multiplication is completed (except when OSBW preconditioning is used).

As for (2), each $\mathbf{V}_k$ is obtained using standard two-tiered block Jacobi diagonalization as a sequential product of Jacobi rotation matrices, $\mathbf{R}_{j_1,j_2}^k(\phi)$, of the following form [23,28–30]:

$$[\mathbf{R}_{j_1,j_2}^k]_{i_k,i'_k}(\phi) = \begin{cases} \cos\phi & \text{if } i_k = i'_k = j_1 \text{ or } i_k = i'_k = j_2; \\ \sin\phi & \text{if } i_k = j_1 \text{ and } i'_k = j_2; \\ -\sin\phi & \text{if } i_k = j_2 \text{ and } i'_k = j_1; \\ \delta_{i_k,i'_k} & \text{otherwise.} \end{cases} \tag{9}$$

The sequential matrix–vector products themselves are computationally inexpensive. However, determination of the optimal rotation angles, ϕ, requires $\mathbf{H}_k^{\mathrm{O}}$ matrix elements gathered over all values of the $l < k$ indices, i_l and i'_l [23,28–30]. In light of Section 3.2, this in turn requires parallel communication if $k > s$, though only among the n^{k-s} nodes of a given block group. The most expensive communication is associated with the outermost level d, for which collective gather operations over the full cluster are required.

An important issue in parallelization is load balancing. The regularity of the operations described above ensures that essentially perfect load balancing is achieved for the $p = n^{d-s}$ case considered here, and in the more general case where $n = n_k$ varies with k, provided p is equal to some product of the n_k's, or their factors. For the present pedagogical application, it is convenient to be able to rule out load balancing as a source of parallel inefficiency; however, future implementations will be generalized to allow any desired value for p.

4. Numerical results

In this section, we apply the parallel recursive block Jacobi diagonalization algorithm, described in Sections 2 and 3, to a scalable prototype molecular system consisting of d coupled isotropic harmonic oscillators, described by the potential energy function

$$V(x^1,\ldots,x^d) = \sum_{k=1}^{d}(x^k)^2 + 0.1\sum_{i=1}^{d-1}x^d x^i. \tag{10}$$

There are $n = 8$ grid points per dimension, placed at $x_{i_k}^k = (1/2)i_k - 9/4$. Using sinc-DVR discretization [12], the one-dimensional kinetic energy matrices are found to be

$$[\mathbf{h}_k]_{i_k,i'_k} = (-1)^{i_k-i'_k} 2 \begin{cases} \pi^2/3 & \text{if } i_k = i'_k; \\ 2/(i_k - i'_k)^2 & \text{otherwise.} \end{cases} \tag{11}$$

Eqs. (11), (A.2) and (A.4) are then used to construct the Hamiltonian matrix **H**.

Two types of scalability studies were performed. Data/system size scalability was investigated by varying d while keeping $n = 8$ fixed. Parallel scalability was investigated by varying p (or equivalently, s). Finally, various dimensional combination possibilities were investigated (Section 4.3). The code was written in FORTRAN 90 and MPI, and all numerical tests were performed on the Jazz platform [36,37] – a Linux-based PC computing cluster with 350 2.4 GHz Pentium Xeon single-CPU compute nodes, networked via Myrinet 2000.

4.1. Data scalability for a single compute node

The data scalability study was performed using the same parallel codes as in Section 4.2, except that the number of nodes p was specified to be one. This experiment serves to verify whether CPU time scales linearly with number of operations for a single CPU, as expected. It also serves as a benchmark for the Section 4.2 parallel scalability study. All d values from $d = 3$ to $d = 7$ were considered (the $d = 8$ case exceeds the physical memory limits for a single node [36]). For varying d and constant n, the $\mathcal{O}(n^{d+2})$ scaling leads to

$$\log(\#\,\text{ops}) \propto \log N + \text{const.} \tag{12}$$

Fig. 5 is a log–log plot of CPU time as a function of N. From the figure, it is clear that the expected scaling relationship is indeed observed, for all data sizes except the smallest considered ($d = 3$). In general, the CPU times involved are quite fast, even for large d.

4.2. Parallel scalability: speedup and efficiency

Two natural indicators of parallel scalability are *speedup* and *efficiency*. Speedup is defined as the CPU time-to-solution of a given task as performed on a single CPU, divided by that of the same task performed on multiple CPUs. Parallel efficiency is speedup divided by p. When comparing different speedup and efficiency values to evaluate scalability, the obvious question one first encounters is, how should the system size scale with p? One can keep the system size constant, in which case unrealistic efficiencies are obtained in the large p limit. A better approach would be to increase the system size with p such that the memory per CPU remains constant. For linear algebra applications, one might argue that N or N^2 should be proportional to p. More sophisticated approaches have also been suggested, such as the isoefficiency scalability method [38–40], discussed below.

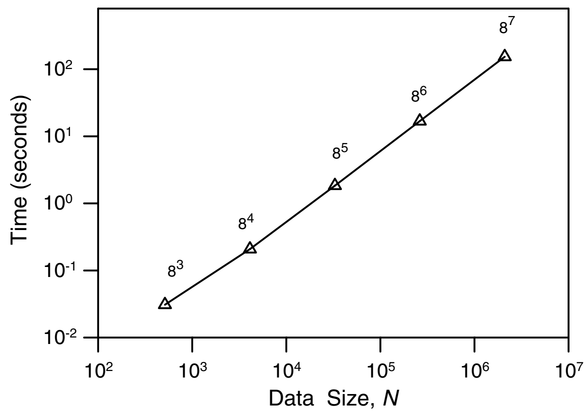


Fig. 5. CPU time as a function of data size, for recursive block Jacobi diagonalization as performed using parallel codes on a single node ($p = 1$). Data size is given by $N = 8^d$, where d is system dimensionality. All values from $d = 3$ to $d = 7$ are represented.

In this section, the parallel scalability of block Jacobi diagonalization is investigated, vis-a-vis parallel speedup and efficiency. Regarding the issue of system size vs. number of processors, our initial approach will simply be to vary both parameters, i.e. d and p, simultaneously. The former now varies from $d = 4$ to $d = 8$ (i.e. $N = 8^8 \approx 2 \times 10^7$ for the largest case). As for p, since $n = 8 = 2^3$, we expect perfect load balancing to be achieved whenever p is any power of two. This restriction will be presumed for simplicity, and in order that we may definitively rule out load balancing as a source of inefficiency for the present study. The following p values were thus considered: $p = 1$, 2, 4, 8, 16, 32, 64 and 128.

The speedup and parallel efficiency are shown in Figs. 6(a) and (b), respectively. For all plots except $d = 8$, speedup values are obtained via comparison with the $p = 1$ results from Section 4.1. For $d = 8$, the $p = 1$ calculation cannot be performed, as discussed in the previous subsection. Consequently, the speedup is first defined relative to the corresponding $p = 4$ calculation, and then multiplied by a typical speedup value for $p = 4$, e.g. 4.0 in this case. From the figures, it is clear that speedup and parallel efficiency increase with data size for fixed p, as expected. For the block Jacobi case with $p = 64$ nodes, for example, the parallel efficiency is 21% for $d = 4$ ($N \approx 4000$), 69% for $d = 5$ ($N \approx 32{,}000$), and nearly 100% for $d = 6$ ($N \approx 256{,}000$) and above. For $d = 6$ and above, the speedup is nearly linear with p even up to 128 nodes. These results are extremely encouraging vis-a-vis parallel scalability, particularly when extrapolated to larger calculations. To this end, an isoefficiency analysis can be applied, as follows.

Imagine scaling N with p such that the memory requirements per CPU were to remain constant. For the present application, this corresponds to linear scaling, i.e., to $N \propto p$. Thus, provided that the efficiency remains constant, or increases with increasing N or p, then the algorithm should be scalable indefinitely, i.e. up to arbitrarily large p. Conversely, if N is instead scaled with p so as to maintain constant *efficiency*, then indefinite

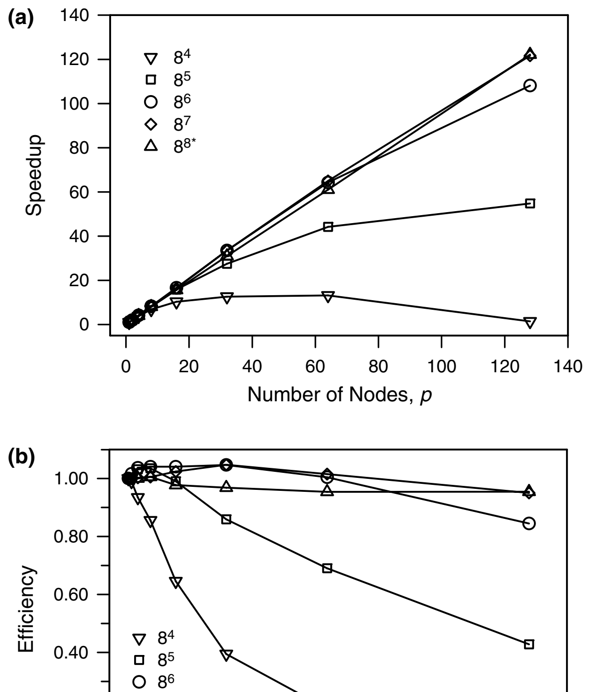


Fig. 6. Speedup (a) and parallel efficiency (b) of parallel recursive block Jacobi diagonalization, as a function of number of nodes, p, for various data sizes $N = 8^d$, ranging from $d = 4$ to $d = 8$.

scaling requires that N increases linearly with p, or less quickly. From Fig. 6(b), there are three calculations with parallel efficiency values in the vicinity of 0.83. These values suggest that the isoefficient scaling of N vs. p for the block Jacobi operation scales as $N \propto p^{3/2}$ – i.e., the memory per CPU scales as $p^{1/2}$. Although not indicative of completely indefinite scaling, this nevertheless represents a very modest increase in per-CPU memory requirements. Table 1 provides an estimate of the number of matrix entries that must be stored per CPU as a function of p, in order to maintain 0.83 efficiency. This includes the actual values for the $p \leqslant 128$ calculations performed here, as well as projections for larger p values. Since the $p \leqslant 128$ memory requirements are not very large to begin with, it is clear that the block Jacobi algorithm should be scalable to extremely large clusters indeed, before memory or efficiency limitations are encountered.

4.3. Dimensional combination

Although the results of Section 4.2 indicate very favorable parallel scalability, it will be shown in paper II that the basic matrix–vector product procedure does not parallelize nearly so well. The basic reason is that the number of operations scales only as n^{d+1} rather than n^{d+2} so that the procedure is communication-intensive rather than CPU-intensive. A reasonable strategy, therefore, is to increase the ratio of CPU-to-communication times, by somehow artificially reducing the sparsity of the matrices involved. We have already discussed one approach that achieves this automatically, i.e. the standard two-tiered block Jacobi diagonalization of Section 2, as applied to systems with $d > 2$. This approach divides up all of the d dimensions into two categories, inner and outer. From a numerical point of view, the resultant matrices behave as if all of the dimensions within the same category were combined together into a single dimension, so that the resultant sparsity pattern corresponds to $d = 2$ and Fig. 2(a). In effect, d is greatly reduced, but since N must still be the same, the effective n value is greatly increased.

In light of the recursive generalization of the block Jacobi procedure developed in Section 2, there is no reason why we cannot allow completely arbitrary partitionings of dimensions into separate categories, each of which constitutes a single "effective" dimension. In other words, the indices $i_{1 \leqslant k \leqslant d}$ are replaced with effective dimension indices $i'_{1 \leqslant k' \leqslant d'}$, where

$$\begin{aligned} i'_1 &= (i_1, i_2, \ldots, i_{l_1}), \\ i'_2 &= (i_{l_1+1}, \ldots, i_{l_2}), \\ &\vdots \\ i'_{d'} &= (i_{l_{d'-1}+1}, \ldots, i_d). \end{aligned} \tag{13}$$

Indeed, there are often physical motivations for such a "dimensional combination" procedure. For example, it might be advantageous to lump together two strongly coupled dimensions [27], or the three Cartesian vector components that describe a single atom. For non-Cartesian applications of block Jacobi diagonalization, there are certain situations where dimensional combination *must* be applied [14,30,41].

Dimensional combination offers other potential advantages as well (paper II). For the present purpose, however, we will be concerned only with its effects on parallel scalability. The primary effect is to reduce

Table 1
Isoefficiency scalability study for parallel block Jacobi diagonalization algorithm

System dimensionality d	Grid size N	Number of nodes p	Data per node $8^{d+1}/p$
4	4096	8	4096
5	32,768	32	8192
6	262,144	128	16,384
7	2,097,152	512	32,768
8	16,777,216	2048	65,536
9	134,217,728	8192	131,072
10	1,073,741,824	32,768	262,144

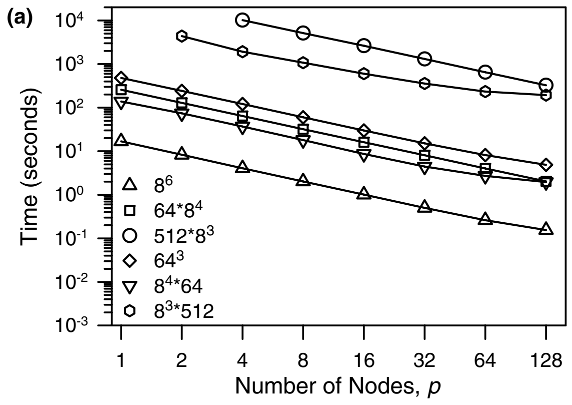
For each system dimensionality, d, column III lists the number of nodes, p, for which the calculation achieves a parallel efficiency of 0.83. The corresponding amount of data (number of distinct matrix entries) which must be stored per node is given in column IV.

the sparsity of the matrices involved. The original **H** matrix is of course the same, but the $\mathbf{H}^{D}_{1'}$, $\mathbf{H}^{O}_{kl'}$, and $\mathbf{V}_{k'}$ matrices that result from block Jacobi diagonalization are now completely different. As per Section 2, the number of CPU operations required to perform block Jacobi diagonalization increases substantially, as does that of the resultant matrix–vector products.

For the numerical investigation conducted here, six different configurations of dimensional combinations were considered, all applied to the same initial $d = 6$ system. An example will serve to clarify the notation used: "$64 * 8^4$" $= (8 * 8) * 8 * 8 * 8 * 8$ implies that the innermost two dimensions have been combined into one effective dimension (i.e. $l_1 = 2$), etc. The six configurations are as follows: 8^6; $64 * 8^4$; $512 * 8^3$; 64^3; $8^4 * 64$; and $8^3 * 512$. As in Section 4.2, all power-of-two values of p up to $p = 128$ were considered. Total computational time and speedup data for the resultant block Jacobi diagonalizations are presented in Figs. 7(a) and (b), respectively.

From (a), it is clear that dimensional combination increases the CPU time required for block Jacobi diagonalization very substantially. This is to be expected – the parallel efficiency for the uncombined 8^6 case is already very good, and cannot be significantly improved. We also find a large jump in CPU time between configurations that combine dimensions in groups of two (i.e. the "64 configurations") vs. groups of three (the "512" configurations). This is also to be expected; according to the scaling described in Section 2, the number of operations increases by roughly a factor of $n^2 = 64$, for each dimension in the largest category. This is comparable to, but somewhat worse than, the scaling that is actually observed in Fig. 7(a).

The figure also indicates significant configurational "splitting" within the 64 and 512 categories. Configurations with multiple dimensional combinations are somewhat more expensive than those with only a single dimensional combination of the same size, which is certainly to be expected. More surprising, perhaps, is the fact that combining outer dimensions is significantly less expensive than combining inner dimensions insofar as block Jacobi diagonalization is concerned. The memory requirements are also somewhat reduced. On the



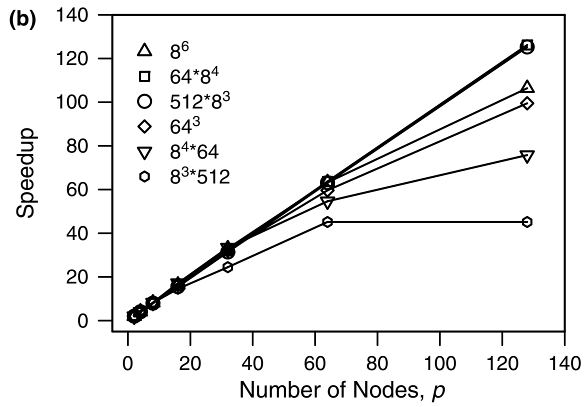


Fig. 7. CPU time (a) and speedup (b) of parallel recursive block Jacobi diagonalization, as a function of number of nodes, p, for $d = 6$ system with various dimensional combination schemes.

other hand, the parallel efficiency for large p is substantially worse for combined outer dimensions than for combined inner dimensions, as is evident in Fig. 7(b) – indeed, the outer case is even worse than for *no* dimensional combination. Despite this, and the large relative increases in CPU times when dimensions are combined, the absolute block Jacobi diagonalization times are still quite small, and unlikely to be the bottleneck of the total preprocessing-plus-iterative-solve operations required to solve the total eigenproblem or linear solve problem. As discussed in paper II, this is because the cost reduction for the iterative solve portion may far outweigh the additional preprocessing cost paid here.

Acknowledgments

This work was largely supported by the Office of Advanced Scientific Computing Research, Mathematical, Information, and Computational Sciences Division of the US Department of Energy under contract DE-FG03-02ER25534. Additional support from The Welch Foundation (D-1523) is also acknowledged. The authors wish to express special gratitude to Michael Minkoff and Albert F. Wagner, whose interest and motivation have made this work possible. Other researchers, notably Tucker Carrington, Jr., Stephen K. Gray, Dinesh K. Kaushik, Dmitry M. Medvedev, Ron Shepard, and Barry F. Smith, are also acknowledged for many stimulating discussions. In addition, we gratefully acknowledge use of "Jazz", a 350-node computing cluster operated by the Mathematics and Computer Science Division at Argonne National Laboratory as part of its Laboratory Computing Resource Center.

Appendix A. Structured grids and the A matrix form

Consider the two-dimensional Cartesian Hamiltonian,

$$\hat{H} = -\frac{\hbar^2}{2m}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) + V(x,y), \tag{A.1}$$

and the matrix representation $\mathbf{H}$ obtained from the $n \times n$ structured rectilinear grid of points (x_i, y_j). As indicated schematically in Fig. 1, the grid points need not be spaced uniformly. Individual matrix elements are specified using both row indices (i,j) and column indices (i',j'). In most discretization schemes, the potential is represented as a diagonal matrix,

$$\mathbf{\Delta}_{i,j,i',j'} = \delta_{ii'}\delta_{jj'}V(x_i, y_j). \tag{A.2}$$

The schemes differ with respect to how they represent the (Laplacian) kinetic energy contribution, but in all cases, this involves off-diagonal coupling along a single dimension at a time. Thus, if $\mathbf{h}$ is the one-dimensional kinetic energy matrix according to some particular discretization scheme, then

$$\mathbf{H} = \mathbf{h}_x \otimes \mathbf{I}_y + \mathbf{I}_x \otimes \mathbf{h}_y + \mathbf{\Delta}. \tag{A.3}$$

If the ordering of grid points is as indicated in Fig. 1 (lexicographical in x then y), then $\mathbf{H}$ will adopt a natural block structure, for which (i,i') labels a matrix element *within* a block, and (j,j') labels an individual block. For this reason, we refer to x and y as "inner" and "outer" dimensions, respectively. The sparsity pattern of $\mathbf{H}$ arises mainly from the kinetic energy terms. In particular, $\mathbf{h}_x \otimes \mathbf{I}_y$ is "block-diagonal", meaning that only the diagonal blocks have non-zero matrix elements, and $\mathbf{h}_x \otimes \mathbf{I}_y$ is "diagonal-block", meaning that all blocks are themselves diagonal. The sum of all three terms in Eq. (A.3) thus matches Fig. 2(a).

All of the above may be generalized for arbitrary dimensionalities d, with dimensions x^k and indices i_k (row) and i'_k (column). In this context,

$$\mathbf{H} = \mathbf{h}_1 \otimes \mathbf{I}_2 \otimes \cdots \otimes \mathbf{I}_d + \mathbf{I}_1 \otimes \mathbf{h}_2 \otimes \cdots \otimes \mathbf{I}_d + \cdots + \mathbf{I}_1 \otimes \mathbf{I}_2 \otimes \cdots \otimes \mathbf{h}_d + \mathbf{\Delta}, \tag{A.4}$$

and lexicographical ordering leads to the self-similar recursive block sparsity pattern of Fig. 2(b). Assuming n grid points per dimension, since the $\mathbf{h}_k$ are in general dense matrices, the number of non-zero matrix elements scales as n^{d+1}. Many further generalizations are also possible, e.g. for effective potentials, non-Cartesian coordinate systems, non-direct-product basis sets, etc. [23,28–30], but these will not be considered here.

References

[1] B. Poirier, T. Carrington Jr., J. Chem. Phys. 118 (2003) 17.
[2] J.G. Muga, J.P. Palao, B. Navarro, I.L. Egusquiza, Phys. Rep. 395 (2004) 357.
[3] W.H. Miller, J. Chem. Phys. 62 (1975) 1899.
[4] T. Seideman, W.H. Miller, J. Chem. Phys. 96 (1992) 4412.
[5] U. Manthe, W.H. Miller, J. Chem. Phys. 99 (1993) 3411.
[6] U. Manthe, T. Seideman, W.H. Miller, J. Chem. Phys. 101 (1994) 4759.
[7] W. Chen, B. Poirier, J. Comput. Phys. (this issue), doi:10.1016/j.jcp.2006.03.031.
[8] G. Strang, Linear Algebra and its Applications, Academic Press, Orlando, FL, 1980.
[9] D.O. Harris, G.G. Engerholm, W.D. Gwinn, J. Chem. Phys. 43 (1965) 1515.
[10] A.S. Dickinson, P.R. Certain, J. Chem. Phys. 49 (1968) 4209.
[11] J.C. Light, I.P. Hamilton, J.V. Lill, J. Chem. Phys. 82 (1985) 1400.
[12] D.T. Colbert, W.H. Miller, J. Chem. Phys. 96 (1992) 1982.
[13] V. Szalay, J. Chem. Phys. 105 (1996) 6940.
[14] J.C. Light, T. Carrington Jr., Adv. Chem. Phys. 114 (2000) 263.
[15] R.G. Littlejohn et al., J. Chem. Phys. 116 (2002) 8691.
[16] J. Echave, D.C. Clary, Chem. Phys. Lett. 190 (1992) 225.
[17] H. Wei, T. Carrington Jr., J. Chem. Phys. 97 (1992) 3029.
[18] M.J. Bramley, T. Carrington Jr., J. Chem. Phys. 99 (1993) 8519.
[19] B. Poirier, J.C. Light, J. Chem. Phys. 111 (1999) 4869.
[20] B. Poirier, Found. Phys. 30 (2000) 1191.
[21] B. Poirier, J.C. Light, J. Chem. Phys. 114 (2001) 6562.
[22] B. Poirier, Found. Phys. 31 (2001) 1581.
[23] B. Poirier, Numer. Linear Algebra Appl. 7 (2000) 715.
[24] R. Geus, S. Röllin, Parallel Computing 27 (2001) 883.
[25] A. George, J.W. Liu, Computer Solution of Large Sparse Positive Definite Matrices, Prentice-Hall, Englewood Cliffs, NJ, 1981.
[26] Y. Saad, Iterative Methods for Sparse Linear Systems, PWS, Boston, 1996.
[27] B. Poirier, T. Carrington Jr., J. Chem. Phys. 114 (2001) 9254.
[28] B. Poirier, W.H. Miller, Chem. Phys. Lett. 265 (1997) 77.
[29] B. Poirier, Phys. Rev. A 56 (1997) 120.
[30] B. Poirier, J. Chem. Phys. 108 (1998) 5216.
[31] B. Poirier, T. Carrington Jr., J. Chem. Phys. 116 (2002) 1215.
[32] W. Bian, B. Poirier, J. Theoret. Comput. Chem. 2 (2003) 583.
[33] W. Bian, B. Poirier, J. Chem. Phys. 121 (2004) 4467.
[34] R.E. Wyatt, Phys. Rev. E 51 (1995) 3643.
[35] B.F. Smith, P. Bjørstad, W. Gropp, Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations, first ed., Cambridge University Press, New York, 1996.
[36] Available from: <http://www.lcrc.anl.gov/jazz/index.php>.
[37] Available from: <http://www.lcrc.anl.gov/jazz/Documentation/Policies/index.php>.
[38] V. Kumar, V.N. Rao, Int. J. Parallel Programming 16 (1987) 501.
[39] A. Grama, A. Gupta, V. Kumar, IEEE Parallel Distrib. Technol. 1 (1993) 12, Special issue on parallel and distributed systems: from theory to practice.
[40] V. Kumar, A. Gupta, J. Parallel Distrib. Computing 22 (1994) 379.
[41] J. Dai, J.C. Light, J. Chem. Phys. 107 (1997) 8432.