Available online at www.sciencedirect.com

ScienceDirect

Journal of Computational Physics 219 (2006) 198–209

JOURNAL OF COMPUTATIONAL PHYSICS

www.elsevier.com/locate/jcp

Parallel implementation of efficient preconditioned linear solver for grid-based applications in chemical physics. II: QMR linear solver

Wenwu Chen *, Bill Poirier

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

Received 6 September 2005; received in revised form 7 March 2006; accepted 18 March 2006
Available online 16 May 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 [B. Poirier, 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 II) indicates that the basic parallel sparse matrix–vector product operation itself is the overall scalability bottleneck, faring much more poorly than the specialized, block Jacobi routines considered in a companion paper (paper I). However, a simple dimensional combination scheme is found to alleviate this difficulty.
© 2006 Elsevier Inc. All rights reserved.

PACS: 33.20.−t; 82.20.−w

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

1. Introduction

This paper (paper II) is the second in a two-part series exploring the parallel scalability of sparse iterative linear algebra calculations for block-structured matrices, such as arise in chemical physics and other scientific and engineering fields. The authors' specific research focus is the quantum dynamics of molecular systems, which has proven to be one of the most challenging computational problems of current interest, motivating the development of scalable algorithms for massively parallel computers. It should be stated clearly at the outset, however, that the methods developed here may be applied to a very broad range of applications, e.g., any multidimensional partial differential equation (PDE) simulation on a structured grid. In all such cases, the

* 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.2006.03.031

matrices involved are characterized by off-diagonal blocks that are diagonal, and diagonal blocks that are either dense, or themselves block-structured as described above. A schematic illustration may be found in paper I [1], and a more precise description of the required matrix form is given in Ref. [2].

The focus of this paper is the matrix–vector product operation itself, and the associated iterative linear solver routines [3]. The basic matrix–vector product operation is one of the most fundamental numerical operations imaginable, and it is hardly surprising that various parallel sparse matrix–vector product algorithms have arisen in recent years [4,5]. However, *none* of these scale effectively for general matrices of the above form (denoted here as '**A** matrices'), which may be regarded as a serious deficiency. Indeed, one previous study [4] deemed the parallel performance for **A**-type test matrices to be "unacceptably low", observing speedups of only around 2×, even when 64 nodes were used.

That the general **A** matrix constitutes a "worst-case scenario" for parallel sparse matrix–vector products is readily apparent upon consideration of the most commonly used implementation, i.e. compressed sparse row (CSR) storage of the non-zero matrix elements [4,5], and subsequent parallel distribution of matrix data across nodes by contiguous groups of rows ("block-row distribution"). Under such a scheme, relatively little inter-node communication is required to implement the matrix–vector product, *provided* that the profile of the matrix, $p(\mathbf{A})$, is much smaller than N [4], where N is the matrix size, and

$$p(\mathbf{A}) = \max(|i-j|) \quad \text{for all } A_{ij} \neq 0. \tag{1}$$

For general **A** matrices, however, $p(\mathbf{A}) \approx N$, thus resulting in communication between virtually *all* nodes, and consequent poor scalability. To remedy this situation, the better parallel algorithms first employ matrix reordering techniques, such as reverse Cuthill–McKee [6], in an attempt to minimize $p(\mathbf{A})$. It can be shown, however [2], that *no* matrix reordering of a general **A** matrix will substantially reduce its profile, no matter how sparse the matrix.

A completely new approach to the parallel sparse matrix–vector problem is therefore introduced in this paper – the first, to our knowledge, to provide "acceptable" scaling for general **A** matrices. Since our primary interest is in solving the linear system $\mathbf{Aw} = \mathbf{v}$, we also discuss the parallel implementation of the most popular iterative linear solver routine, QMR [3]. The basic symmetric version of QMR as presented in Appendix A requires one **A** matrix–vector product per iteration. However, *preconditioned* QMR [1] also requires additional matrix–vector products to be performed each iteration; thus, preconditioning is also addressed here. Note that the *construction* of the preconditioner matrices occurs in a different preprocessing step, addressed separately in paper I. A range of preconditioners are considered, all of which are based on the same block Jacobi diagonalization construction (the preprocessing step), but differ with respect to the subsequent matrix–vector product implementation (the QMR step). Quite apart from issues of parallelization, the block Jacobi family of preconditioners are employed because, for **A** matrices, these have been found to be the most effective by far at reducing the total number of required QMR iterations.

The organization of this paper is as follows. Section 2 overviews the recursive implementation of the **A** and preconditioner matrix–vector products, and the associated QMR linear solver routine, on a single node. As in paper I, a recursive implementation must be developed in order to handle the case where the diagonal blocks have their own subblock structure – corresponding to a physical system with arbitrary dimensionality (the number of layers of block structure in the matrix **A** is equal to the system dimensionality, d). Section 3 then describes the parallel implementation, addressing primarily the data distribution scheme. Section 4 applies the parallel codes to the model system described in paper I. 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, the dimensional combination technique is introduced in Section 4.3. Finally, Section 5 summarizes the current parallel implementation as discussed in both papers, and suggests directions for future development.

2. Iterative linear solvers and preconditioners

2.1. Introduction

As discussed in paper I, the molecular Hamiltonian is represented as a symmetric sparse block matrix, **H**, of the **A** matrix form described in Section 1. There are two basic linear algebra operations to be performed:

(1) the symmetric eigenvalue/eigenvector problem, $(\mathbf{H}-\lambda\mathbf{I})\mathbf{x}=0$; (2) the linear solve problem, $\mathbf{w}=(\mathbf{H}-\lambda\mathbf{I})^{-1}\mathbf{v}$. Our interest in iterative methods [3,7,8] therefore spans both eigensolvers and linear solvers. In practice, however, the former can be converted into the latter via the PIST method [9–11]. Similarly, the SPAM [12] and Davidson [13] iterative eigensolver methods are automatically amenable to block Jacobi preconditioners, at least in the OSB representation (defined in paper I). Consequently, we focus exclusively on iterative linear solvers and preconditioners.

The most well-known linear solvers are the generalized minimum residual (GMRES) [7] and quasi-minimal residual (QMR) [8]. QMR is the more efficient when the required number of iterations M is sizeable – as is often the case in quantum applications when λ is large (paper I). Preconditioning can reduce M substantially, provided that an approximation $\mathbf{P}$ to the matrix $(\mathbf{H}-\lambda\mathbf{I})$ be found, such that matrix–vector products with $\mathbf{P}^{-1}$ are computationally inexpensive. Each QMR iteration then requires one matrix–vector product with both $(\mathbf{H}-\lambda\mathbf{I})$ and $\mathbf{P}^{-1}$ (Appendix A). In chemical physics, the emphasis has primarily been on approximate Hamiltonian preconditioners (AHPs) $\mathbf{P}=(\mathbf{H}_0-\lambda\mathbf{I})$ where $\mathbf{H}_0\approx\mathbf{H}$ [10].

2.2. Recursive matrix–vector product implementation for OSB preconditioning

The QMR algorithm requires scalar, vector, and matrix operations (Appendix A). As the first two are completely straightforward (even in parallel) our primary task is to implement the two matrix–vector products, $(\mathbf{H}-\lambda\mathbf{I})\mathbf{x}$ and $\mathbf{P}^{-1}$ $\mathbf{x}$, for systems of arbitrary dimensionality. For single-node environments, a direct implementation of $(\mathbf{H}-\lambda\mathbf{I})$ $\mathbf{x}$ (or just $\mathbf{Hx}$) is also straightforward (e.g., via the standard CSR approach), because the memory required to store $\mathbf{H}$ scales only as $n^d=N$, where n is the number of grid points per dimension. On the other hand, $\mathbf{P}^{-1}\mathbf{x}$ requires some consideration, owing to the atypical matrix storage scheme resulting from the recursive block Jacobi procedure of paper I.

Note that recursive block Jacobi leads to a natural choice of AHP, i.e. the diagonal matrix $\mathbf{P}=(\mathbf{H}_1^{\mathrm{D}}-\lambda\mathbf{I})$, because

$$||\mathbf{H}_{\mathrm{OSB}}-\mathbf{H}_1^{\mathrm{D}}||\ll||\mathbf{H}_1^{\mathrm{D}}||, \tag{2}$$

(where $\mathbf{H}_{\mathrm{OSB}}$ is the OSB representation of $\mathbf{H}$), and the matrix–vector product $\mathbf{P}^{-1}\mathbf{v}$ is computationally inexpensive to compute. This is known as the "OSB preconditioner" [2,14–16], the simplest and computationally least expensive of the block Jacobi family. The cost to compute and store $\mathbf{P}^{-1}$ from $\mathbf{H}_1^{\mathrm{D}}$ is trivial, and need only be performed a single time, in the preprocessing step. However, the $\mathbf{P}^{-1}$ basis representation (OSB) is completely different from that of the QMR vectors. Consequently, $\mathbf{P}^{-1}$ cannot be applied directly during each QMR iteration, but must instead be transformed to the original grid representation via the orthogonal transformation

$$\mathbf{P}_{\mathrm{GRID}}^{-1}=\mathbf{V}_d\mathbf{V}_{d-1}\cdots\mathbf{V}_2\mathbf{V}_1(\mathbf{H}_1^{\mathrm{D}}-\lambda\mathbf{I})^{-1}\mathbf{V}_1^{\mathrm{T}}\mathbf{V}_2^{\mathrm{T}}\cdots\mathbf{V}_{d-1}^{\mathrm{T}}\mathbf{V}_d^{\mathrm{T}}, \tag{3}$$

where the sparse, block-diagonal transformation matrices, $\mathbf{V}_k$ (Fig. 2), are computed and stored during the block Jacobi procedure. The index $k\leqslant d$ labels a particular system dimension, or equivalently, a level within the recursive $\mathbf{A}$ matrix block hierarchy [1]. The definition of the $\mathbf{V}_k$, and their relation to the partially transformed block-diagonal and off-block-diagonal Hamiltonian matrices, $\mathbf{H}_k^{\mathrm{D}}$ and $\mathbf{H}_k^{\mathrm{O}}$, are discussed in paper I. The CPU cost for each of the individual matrix–vector products implicit in Eq. (3) scales as n^{d+1}, which is identical to that of the $\mathbf{Hx}$ product. Note that no Hamiltonian matrices other than $\mathbf{H}$ and $\mathbf{H}_1^{\mathrm{D}}$ need be stored in order to implement the necessary matrix–vector products.

2.3. Improved block Jacobi preconditioners

The OSB preconditioner described above works very well in most molecular applications [2,14–16] vis-a-vis greatly reducing the value of M. Like all AHPs, however, M still tends to increase dramatically with λ, owing to the increasing spectral density manifesting as large diagonal matrix elements for $(\mathbf{H}_1^{\mathrm{D}}-\lambda\mathbf{I})^{-1}$ (when the $\mathbf{H}_1^{\mathrm{D}}$ elements are in the vicinity of λ). By computing some of the off-diagonal matrix elements of $\mathbf{H}_{\mathrm{OSB}}$ in the vicinity of λ, and incorporating those into $\mathbf{P}$, the situation can often be greatly improved, to the extent that M becomes very small throughout the spectral range of $\mathbf{H}$ [10,11,17]. This is called "OSBW preconditioning".

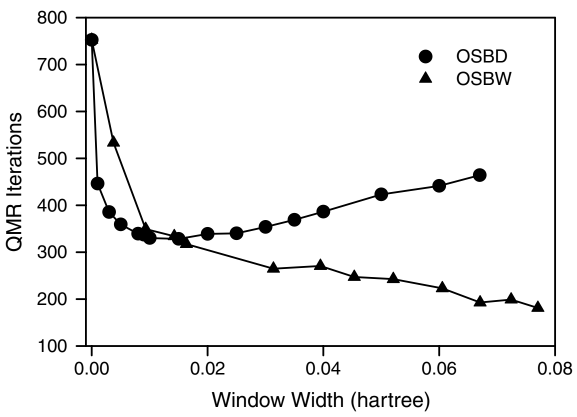

Fig. 1. Number of QMR iterations, M, vs. window width, $2x_0$, for OSBD and OSBW preconditioners, as applied to a calculation of the vibrational states of acetylene. Energy is $\lambda = 0.055$ hartree.

Unfortunately, the computation of the additional off-diagonal matrix elements (using Eq. (7) from paper I) appears to be ill-suited to parallelization, according to preliminary numerical investigations.

A compromise solution is the OSB plus diagonal, or "OSBD" preconditioner, introduced here for the first time. Like the OSB preconditioner, the OSBD $\mathbf{P}$ matrix is diagonal in the OSB representation. A completely generic OSBD preconditioner may be defined as $\mathbf{P} = f[(\mathbf{H}_1^{\mathrm{D}} - \lambda\mathbf{I})]$, where $f[x]$ is an arbitrary function. In general, it should be chosen such that $f[x] \approx x$ when $|x|$ is sufficiently far from zero, in order to satisfy the good preconditioner requirement. As $x \to 0$, however, $|f[x]| \gg |x|$, in order to avoid near singularities in $\mathbf{P}^{-1}$. For this paper, we use the particular function

$$f[x] = \begin{cases} x & \text{if } |x| > x_0, \\ x_0(x/|x|) & \text{if } |x| \leqslant x_0, \end{cases} \tag{4}$$

where x_0 is a user-defined parameter. The choice of x_0 will determine the preconditioner efficiency vis-a-vis minimization of M, but will not affect the per iteration CPU cost, nor the cost of preconditioner construction.

Preliminary numerical tests indicate that the OSBD preconditioner efficiency is indeed generally intermediate between that of OSB and of OSBW, as expected, but can be even more efficient than OSBW for small window sizes. Fig. 1 indicates the number of QMR iterations required to compute the vibrational states of acetylene using PIST with OSBD and OSBW preconditioning, as a function of the window width, $2x_0$ (numerical details of this calculation will be provided on request). The energy is $\lambda = 0.055$ hartree. Note that the $\mathbf{x}_0 = 0$ limit corresponds to ordinary OSB preconditioning. Very substantial reductions in M are observed. The OSBD plot exhibits an expected minimum, near 0.01 hartree; however, the efficiency for larger widths is quite insensitive to the particular value of x_0. This is promising from the point of view of being able to apply OSBD to real molecular applications, without undue fussing with parameters. The corresponding plot for OSBW preconditioning decreases monotonically, but the per-iteration cost also increases substantially [10,11,18,19].

3. Parallel implementation

3.1. Data distribution and implementation

Building on the domain decomposition scheme outlined in paper I, the parallel $\mathbf{Hx}$ and $\mathbf{P}^{-1}\mathbf{x}$ matrix–vector products are implemented using a multi-layer approach. The total number of nodes is presumed to be a power of n, i.e. $p = n^{d-s}$, for $0 < s < d$. For the lower, $k \leqslant s$ levels, vector data is partitioned into p subvectors of length n^s, as indicated in the bottom of Fig. 2. The corresponding k-level matrices are block-diagonal, with an integral number (n^{s-k}) of diagonal blocks stored per node [1]. Thus, no communication is required to implement all of the $k \leqslant s$ matrix–vector products.

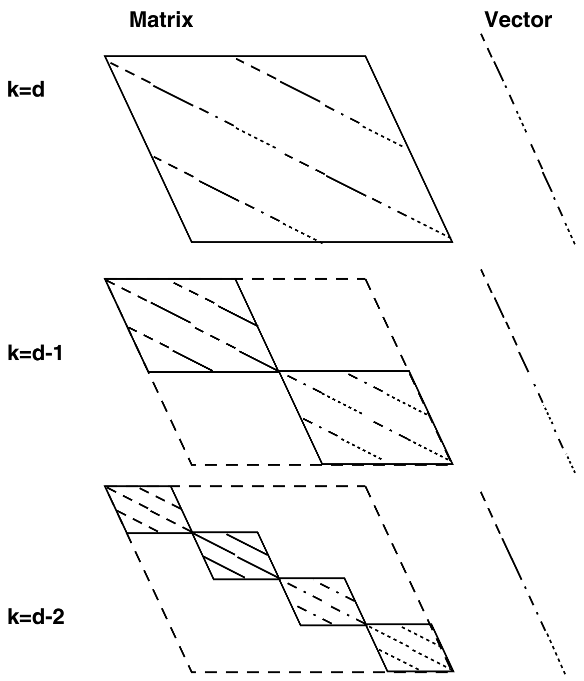

Fig. 2. Schematic of data distribution for matrices and vectors, as employed by the parallel QMR implementation at the highest three dimensional levels, $k = d$, $k = d - 1$, and $k = d - 2$ (with $n = 2$, and $s = d - 2$). At each level, the four different line types (dashed, solid, dot-dashed, and dotted) indicate how data would be distributed across $p = 4$ nodes.

For $k > s$, data is distributed into smaller, non-geographically adjacent pieces, as indicated in the upper parts of Fig. 2. There are now multiple *nodes* per *block* (n^{k-s}), rather than the other way around. No communication *across* blocks is required to implement the matrix–vector product, although the large block profile suggests that much *intra* block communication may be involved. In fact, *all* communication can be avoided, simply by adopting a block-row distribution scheme corresponding to subvectors of length n^{s-1} instead of n^s. Thus, the first n^{s-1} rows of a given 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 again, starting a second cycle. After n complete cycles, each node stores n subvectors (or block rows) of length n^{s-1}.

The above data distribution scheme requires zero communication among nodes to effect the parallel matrix–vector product – a testament to the power of the new domain decomposition (paper I). The drawback, however, is that vector data must be reordered between successive matrix–vector products for different levels k, e.g., when implementing Eq. (3). For k only slightly larger than s, only local parts of the vector need be recombined, i.e. communication occurs within small groups of nodes. At the highest levels, however, i.e. $k \approx d$, the most remote parts of the vector $\mathbf{x}$ must be combined together, requiring collective gather and scatter calls across essentially the entire computing cluster. In effect, the inherent communication difficulty of parallel $\mathbf{A}$ matrix–vector products, though not completely avoided, has been "repackaged" so as to minimize the negative impact on computational efficiency.

3.2. Performance and alternate strategies

Fig. 3 is a detailed schematic indicating how the communication required for parallel vector reordering is implemented at different levels. The compute nodes communicate with the root nodes of their groups, via gather calls to update the output vector from the preceding matrix–vector product operation. The root nodes then scatter the assembled portion of the vector from the previous group to the corresponding group(s) for the next level. As indicated in the figure, group sizes decrease with decreasing k, resulting in smaller assembled

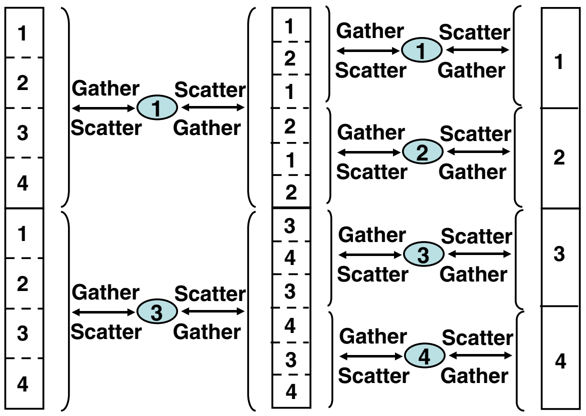

Fig. 3. Schematic of data exchange among nodes, associated with vector reordering that must occur between sequential k-level parallel matrix–vector products. The highest three dimensional levels are indicated, decreasing from left to right. Communication is well distributed among nodes for the lower levels, but increasingly less so with increasing k.

vector portions that must be transferred between any given pair of nodes, and substantially reduced communication times. In contrast, the communication time required by the outermost ($k = d$) level alone can easily come to dominate the computational cost of the whole operation, as has been verified numerically (Section 4). Since this operation must be repeated many times, it is important to find new strategies for combatting this problem. Several successful approaches have already been identified, one of which (dimensional combination [1]) is considered here in Section 4.3.

We also find it useful to discuss briefly two unsuccessful strategies. As an alternative to the above scheme, one might consider using the same vector data distribution for all k levels, thus avoiding reordering altogether. This approach would introduce similar communication difficulties in the k-level matrix–vector products themselves, and would not likely yield any benefit – a conclusion confirmed by numerical tests [20]. We have also investigated whether the differences between the $\mathbf{H}$ and $\mathbf{P}^{-1}$ matrix–vector products might be used to advantage. In particular, the various contributions of $\mathbf{H}$ can be multiplied by $\mathbf{x}$ non-sequentially, and then summed together, whereas the individual matrix–vector products in Eq. (3) must be applied sequentially. However, it is not clear how this could be used to avoid the reordering problem. In any event, the final output vectors must be identically distributed, as they are both utilized together in a given QMR iteration. Thus, any scheme that is found to improve the performance of $\mathbf{Hx}$ must also be applicable to $\mathbf{P}^{-1}\mathbf{x}$ in order to be effective.

Another alternative is to store multiple copies of all of $\mathbf{x}$ on individual nodes, when this is possible. We have explored this data duplication option to some extent, by performing numerical experiments that compare whole vector (data duplication) and partial vector (data distribution) parallel performance. These experiments indicate that the whole vector approach is only slightly more efficient for the $\mathbf{Hx}$ operation than the partial vector approach, and not at all more efficient for the $\mathbf{P}^{-1}\mathbf{x}$ operation. As the whole vector approach does not scale to arbitrarily large N, we do not consider it further. Additional numerical details for all of the studies described qualitatively in this subsection are available on request.

4. Numerical results

In this section, the scalability of the recursive OSB-preconditioned QMR procedure is investigated, via application to the variable-size d-dimensional coupled oscillator problem of paper I ($n = 8$ grid points per dimension, $N = n^d$). As in the block Jacobi case (paper I), both a data scalability study for a single compute node, and a parallel scalability study over many nodes, were conducted. The code was written in FORTRAN 90 and MPI, and all numerical tests performed on the Jazz platform [1,20]. Although in some cases, the $\mathbf{Hx}$ and $\mathbf{P}^{-1}\mathbf{x}$ matrix–vector product operations were investigated separately, the primary emphasis is on the complete parallel QMR iteration. (Note that the value of λ is immaterial.)

4.1. Data scalability for a single compute node

The data scalability study described in this subsection was performed using exactly 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 (for the same reasons as presented in paper I for the block Jacobi case). All d values from $d = 3$ to $d = 7$ are considered.

Fig. 4 is a log–log plot of CPU time as a function of N, for the two matrix–vector product operations, $\mathbf{Hx}$ and $\mathbf{P}^{-1}\mathbf{x}$, as well as for a pair of QMR iterations (used rather than just a single QMR iteration for technical reasons). It is obvious that the expected linear scaling relationship holds for all operations, except for the smallest data size considered ($d = 3$). It is also clear that $\mathbf{P}^{-1}\mathbf{x}$ is somewhat more costly than $\mathbf{Hx}$, by about a factor of two. This is reflected in the fact that the former involves roughly twice as many individual matrix–vector products. The figure also indicates that for the pair of QMR iterations, which requires three $\mathbf{P}^{-1}\mathbf{x}$ operations and two $\mathbf{Hx}$ operations, the majority of the CPU time (~90%) is spent doing matrix–vector products. Finally, we note that the corresponding block Jacobi diagonalization [1] is more than an order of magnitude more time-consuming than a pair of QMR iterations. This does not imply that block Jacobi would be the overall computational bottleneck, because an actual linear solve calculation (not performed here) would require many QMR iterations for a single block Jacobi diagonalization.

4.2. Parallel scalability: speedup and efficiency

Figs. 5(a) and (b), respectively, indicate the speedup and parallel efficiency for parallel QMR applied to various system sizes, $N = 8^4$, 8^5, 8^6,8^7, and 8^8. Except for $N = 8^8$, all values are calculated by comparing CPU times with those for $p = 1$. For $N = 8^8$, the $p = 1$ calculation cannot be performed on a single node due to memory restrictions. In this case, the speedup is calculated relative to the corresponding $p = 4$ value, and multiplied by the estimated QMR speedup at $p = 4$ of 2.5.

Compared with the performance of the block Jacobi diagonalization, the speedup and parallel efficiency for QMR is rather poor, essentially across all values of d and p. The maximum speedup achieved is only around 5, and this only for $d = 8$ and $p \geqslant 32$. For $p \geqslant 16$, the efficiencies are all under 35%. The comparatively poor scalability of QMR is due to the fact that block Jacobi is considerably more CPU-intensive, in that the number of operations scales as n^{d+2} (paper I) rather than n^{d+1}. The resultant scalability trends are thus completely different, although it is still true that increased data sizes improve matters substantially. Ultimately, the poor scalability may be attributed to essentially two causes. One is the large level of sparsity that characterizes the matrices involved, especially for large d. If the sparsity level were reduced, then the ratio of computation to communication time would increase substantially (i.e. the situation would be more block-Jacobi-like). The

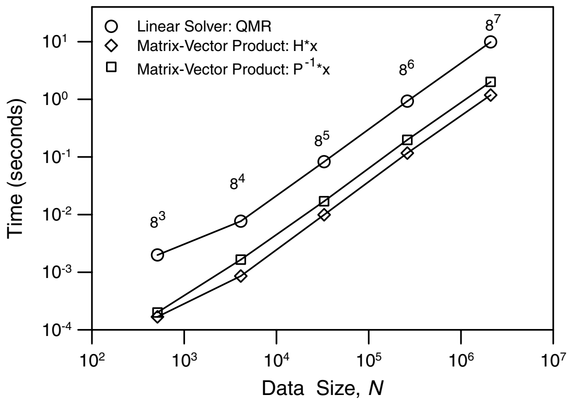

Fig. 4. CPU time as a function of data size, for fundamental operations as performed using parallel codes on a single node ($p = 1$). Data size is given by $N = 8^d$, where d is system dimensionality. All values from $d = 3$ to $d = 7$ are represented. QMR times are for two iterations (including two $\mathbf{Hx}$ and three $\mathbf{P}^{-1}\mathbf{x}$ matrix–vector products).

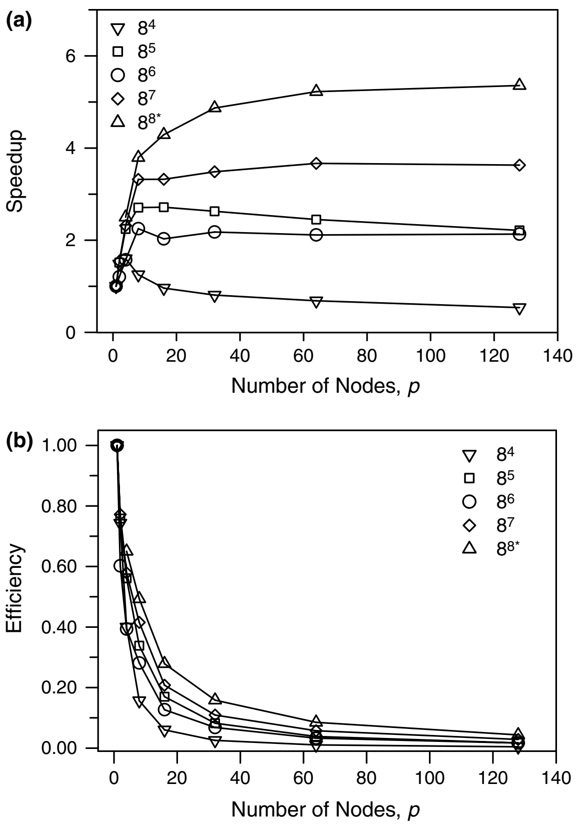

Fig. 5. Speedup (a) and parallel efficiency (b) of two parallel QMR linear solver iterations, as a function of number of nodes, p, for various data sizes $N = 8^d$, ranging from $d = 4$ to $d = 8$.

second reason is the structure of the **A** matrices involved, which necessarily couples together remote regions of the vector, as discussed in Section 3.

4.3. Dimensional combination

The ‘‘unacceptable’’ scaling observed above can be remedied by attacking either of the two causes. In this section, we directly address the sparsity level issue, by applying the dimensional combination scheme of paper I. In effect, this reduces d, while simultaneously increasing n, so that $N = n^d$ remains constant. Since the number of non-zero matrix elements scales as $n^{d+1} = N^{(d+1)/d}$, it is obvious that this will lead to reduced matrix sparsity. This in turn leads to an increased number of CPU operations per matrix–vector product, without directly affecting the vector reordering problem at all. So how is it that such an approach can be beneficial? To start with, the parallel *speedups* are greatly improved, because of the large increase in the ratio of CPU effort to communication, thus greatly mitigating the additional CPU operations when p is large.

There are two additional reasons why dimensional combination is expected to be very beneficial overall. First, the resultant OSB preconditioner is generally much more efficient in the dimensionally combined case, meaning that the total *number* of QMR iterations, M is greatly reduced. This assertion is based on previous work [2,10,11,14–16,18,19], and has nothing to do with parallelization per se. Second, dimensional combination can also be used to reduce the total basis size N without reducing the accuracy of the final calculation [10,11,16,18,19], which leads to fewer overall CPU operations per QMR iteration. At present, however, we concern ourselves only with the case where N remains constant, for simplicity. Note that in this context, dimensional combination increases block Jacobi time [1]; however, if block Jacobi is not the overall computational bottleneck (as is usually the case), then the total time-to-solution will be very substantially reduced.

The optimal amount of dimensional combination is therefore a balance between preprocessing time (block Jacobi) and total QMR linear solve time, and as such depends on the specific application. Our primary goal here is simply to demonstrate numerically that increased dimensional combination does indeed lead to greater parallel scalability, as expected. Figs. 6(a) and (b), respectively, present total computational times and speedups for a pair of QMR iterations for various dimensional combinations of the same $N = 8^6$ problem from Section 4.2. The configurations considered are: 8^6; $64 * 8^4$; $512 * 8^3$; 64^3; $8^4 * 64$; and $8^3 * 512$ (see paper I for explanation of notation). For the number of nodes, p, all power-of-two values up to 128 are considered. From Fig. 6(a), it is evident that the QMR times do not increase nearly so much under dimensional combination as does block Jacobi. This is partly due to the scaling of CPU operations, which only increases by a factor of n rather than n^2, for each incremental increase in the dimension of the largest combined group [1]. As p increases, the observed speedup is far greater for all of the combined configurations than for the uncombined configuration, and the maximum speedup that is now obtained is around 60, instead of 5. In other words, the parallel efficiency is *very substantially improved*, and certainly now within the "acceptable" realm. For the $64 * 8^4$ and $8^4 * 64$ configurations, in fact, the increase in the speedup is so great that by $p = 8$ and beyond, the time per iteration is actually *less* than for the uncombined configuration – even though the number of CPU operations is far greater.

On the other hand, the 64 configurations – representing an intermediate amount of dimensional combination – are only slightly more scalable than the uncombined configuration, in that most of the speedup curves reach their peak by around $p = 32$. Peak speedup values for the 64 configurations are nevertheless several times larger than for the uncombined case. As expected, the best parallel scaling is exhibited by a maximally combined 512 configuration, which is substantially better than all 64 configurations, and has not yet reached maximal speedup even by the largest $p = 128$ value considered. The associated matrix–vector products are also

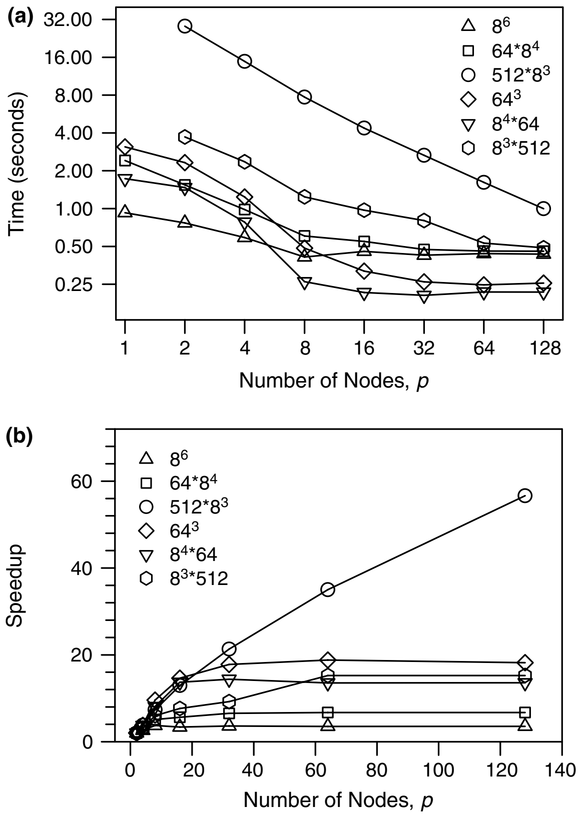

Fig. 6. CPU time (a) and speedup (b) of two parallel QMR linear solver iterations, as a function of number of nodes, p, for $d = 6$ system with various dimensional combination schemes.

the most costly, however, so that by $p = 128$, all configurations require comparable QMR times per iteration. This was the intended result, but suggests that the computational bottleneck is still communication, i.e. the primary practical benefit of dimensional combination will be reduced M and N values. The numerical results are inconclusive with regard to *which* dimensions should be combined for a given level of dimensional combination, i.e. outermost or innermost – as the former is more scalable at the 64 level, but the latter at the 512 level.

5. Summary

Quantum dynamics applications from chemical physics give rise to some of the most numerically challenging PDEs in scientific computing. This is due to the high dimensionalities involved, as well as to large spectral densities, resulting in numerical tasks that are both computation-intensive and memory-intensive. By distributing the computational burden and data among hundreds to tens-of-thousands of nodes, parallel computing promises to enable fully quantum dynamics calculations to be performed for larger polyatomic molecular systems than has heretofore been realized. This will require efficient parallel implementations of both preconditioners and iterative linear solvers, thus motivating the present effort.

In this paper, and the preceding companion paper I [1], we have presented such a parallel implementation, and also performed a detailed scalability study on a reasonably large computing cluster (hundreds of nodes), in order to assess parallel efficiency, and to predict performance on substantially larger tera- and petascale computers (thousands to tens of thousands of nodes). A scalable model Hamiltonian was used for this study which – though not realistic enough to provide sensible results for say, the number of iterations – is nevertheless appropriate for investigating the scalability of single QMR iterations as these might be applied to actual molecular applications.

In contrast to the very high parallel efficiency of the present block Jacobi diagonalization algorithm (i.e. the preprocessing step used to generate all of the present preconditioners) up to the largest number of nodes considered ($p = 128$), the basic QMR iteration procedure generally does not parallelize well beyond $p = 32$ nodes or so, with the bottleneck being the basic sparse matrix–vector product operation. The difficulty appears to be one of bandwidth/data throughput, rather than latency per se, which is very encouraging from the point of view of scalability to much larger clusters, assuming that a solution for smaller clusters can be found. In any event, we have identified the two primary causes for this (Section 4.2), one of which, i.e. the **A** matrix structure, is also the reason why conventional parallel sparse matrix–vector product algorithms fail in such cases. Since **A** matrices arise in all PDE simulations on structured grids, developing efficient matrix–vector product methods for them will undoubtedly be an important area of future hardware and software research, perhaps involving hierarchical memory and/or improved message passing architectures. Indeed, we have already developed one such technique that scales efficiently up to $p = 128$ (even without dimensional combination) as will be reported in a future publication. For the moment, however, the dimensional combination technique of Section 4.3 – which can always be applied in a straightforward manner to any **A** matrix – has also been shown to result in very satisfactory parallel matrix–vector product scaling. Moreover, dimensional combination also yields improved preconditioners, which in turn would reduce the total number of QMR iterations, M, required to perform the linear solve. Dimensional combination is therefore likely to offer substantial numerical improvement for real molecular applications, although a complete assessment would require a detailed study of the effect on M, which we have not provided here.

The work presented here is part of a broader, ongoing effort to develop generic and efficient parallel software to solve exact quantum dynamics problems of various kinds, for polyatomic molecular systems with 3–8 atoms. However, it is hoped that this software may also find safe passage to other disciplines where challenging PDE applications arise. The philosophy underlying the development is similar to that of other national laboratory infrastructure packages such as PETSc [5,21,22]; i.e. that generic, easy-to-use, highly modularized routines are favored over customized code that has been finely tuned for very specific applications. To this end, various versions of the block Jacobi, QMR, $\mathbf{Hx}$, and $\mathbf{P}^{-1}\mathbf{x}$ modules have been implemented, e.g., serial vs. parallel environment, partial vector vs. whole vector, etc. For $\mathbf{P}^{-1}\mathbf{x}$, modules exist for all three preconditioners considered here, i.e. OSB, OSBD, and OSBW (the latter also requires a special version of block Jacobi which retains the off-diagonal $\mathbf{H}_k^{\mathrm{O}}$ matrices). These modules might in principle be incorporated into PETSc, which currently lacks the capability to perform parallel generic **A** matrix–vector products effectively. Although the modules have not yet been officially released, they are currently available from the authors on request.

In addition to the above, many auxiliary modules have also been developed, such as eigensolver modules to implement parallel PIST and Lanczos, and more specialized modules designed to compute quantities specific to quantum dynamics, such as cumulative reaction probabilities [23–26]. Although the modules are generic, they are designed to operate in conjunction with user-specified customization if desired, e.g. specialized grid discretizations, or non-Cartesian coordinate systems. Of course, the basic modules as presented here also show substantial room for future improvement. For instance, many advanced features have been implemented in the new version of MPICH [27], such as parallel IO, remote memory access (RMA) and global array (GA) [28]. The latter in particular – although it does not parallelize data on its own, and therefore cannot be used directly to achieve the functionality described here – nevertheless provides some low-level routines that might be fruitfully applied to the vector-reordering problem in future implementations.

Taking advantage of these advanced features will definitely improve the efficiency of the parallel implementation. Additionally, parallel computing technology is moving towards hybrid structures in between message-passage and shared-memory architectures. Making full use of shared-memory will certainly improve parallel performance on platforms where it is relevant, such as the Seaborg facility at NERSC [29]. In any event, the existing codes can and are already being applied to real molecular applications of topical interest. In particular, we are currently performing fully quantum dynamics calculations to compute the isomerization of acetylene/vinylidene, and the rovibrational spectra of rare gas trimers and tetramers, as will be reported in future publications.

Acknowledgements

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. Simplified QMR algorithm

Below we present a simplified QMR algorithm that can be used to solve the set of linear equations, $\mathbf{Hx} = \mathbf{b}$, when $\mathbf{H}$ is symmetric, and left preconditioning with the inverse preconditioner matrix, $\mathbf{P}$ is applied. The notation differs slightly from that of the paper, i.e. $\mathbf{x}$ now represents $\mathbf{w}$, $\mathbf{b}$ represents $\mathbf{v}$, $\mathbf{H}$ represents $(\mathbf{H} - \lambda\mathbf{I})$, and $\mathbf{P}$ represents $\mathbf{P}^{-1}$.

Simplified QMR algorithm:

```
$\mathbf{x}^{(0)} = 0$;   $\mathbf{r}^{(0)} = \mathbf{v}^{(1)} = \mathbf{b}$;   $\rho_1 = \|\mathbf{b}\|_2$;   $\mathbf{z}^{(1)} = \mathbf{Pb}$;   $\xi_1 = \|\mathbf{z}^{(1)}\|_2$
$\gamma_0 = 1$;   $\eta_0 = -1$;   $\alpha_1 = 1$
DO $i = 1,2,\ldots$
  IF ($\rho_i == 0$ or $\xi_i == 0$) THEN *method fails*
  $\alpha_{i+1} = \alpha_i\xi_i/\rho_i$;   $\mathbf{v}^{(i)} = \mathbf{v}^{(i)}/\rho_i$;   $\mathbf{z}^{(i)} = \mathbf{z}^{(i)}/\xi_i$;   $\delta_i = \mathbf{v}^{(i)} \cdot \mathbf{z}^{(i)}$
  IF ($\delta_i == 0$) THEN *method fails*
  IF ($i == 1$) THEN
    $\mathbf{q}^{(1)} = \mathbf{z}^{(1)}$
  ELSE
    $\mathbf{q}^{(i)} = \mathbf{z}^{(i)} - (\rho_i\delta_i/\epsilon_{i-1})\mathbf{q}^{(i-1)}$
```

```
        END IF
        $\mathbf{p}^{(i)} = \alpha_{i+1}\mathbf{H}\mathbf{q}^{(i)}$;   $\epsilon_i = \mathbf{q}^{(i)} \cdot \mathbf{p}^{(i)}$
        IF ($\epsilon_i == 0$) THEN *method fails*
        $\beta_i = \epsilon_i/\delta_i$
        $\mathbf{v}^{(i+1)} = \mathbf{p}^{(i)} - \beta_i\, \mathbf{v}^{(i)}$;   $\rho_{i+1} = \|\mathbf{v}^{(i+1)}\|_2$
        $\mathbf{z}^{(i+1)} = \mathbf{P}\mathbf{v}^{(i+1)}/\alpha_{i+1}$;   $\xi_{i+1} = \|\mathbf{z}^{(i+1)}\|_2$
        $\theta_i = \rho_{i+1}/(\gamma_{i-1}|\beta_i|)$;   $\gamma_i = 1/\sqrt{1+\theta_i^2}$
        IF ($\gamma_i == 0$) THEN *method fails*
        $\eta_i = -\eta_{i-1}\rho_i\gamma_i^2/(\beta_i\gamma_{i-1}^2)$
        IF ($i == 1$) THEN
            $\Delta_{\mathbf{x}}^{(1)} = \eta_1\alpha_2\mathbf{q}^{(1)}$;   $\Delta_{\mathbf{r}}^{(1)} = \eta_1\mathbf{p}^{(1)}$
        ELSE
            $\Delta_{\mathbf{x}}^{(i)} = \eta_i\alpha_{i+1}\mathbf{q}^{(i)} + (\theta_{i-1}\gamma_i)^2\Delta_{\mathbf{x}}^{(i-1)}$;   $\Delta_{\mathbf{r}}^{(i)} = \eta_i\mathbf{p}^{(i)} + (\theta_{i-1}\gamma_i)^2\Delta_{\mathbf{r}}^{(i-1)}$
        END IF
        $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \Delta_{\mathbf{x}}^{(i)}$;   $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \Delta_{\mathbf{r}}^{(i)}$
        *check convergence, continue if necessary*
    END DO
```

References

[1] W. Chen, B. Poirier, J. Comput. Phys. this issue, doi:10.1016/j.jcp.2006.04.012.
[2] B. Poirier, Numer. Linear Algebra Appl. 7 (2000) 715.
[3] J.J. Dongarra, I.S. Duff, D.C. Sorensen, H.A. van der Vorst, Numerical Linear Algebra for High-Performance Computers, SIAM, Philadelphia, 1998.
[4] R. Geus, S. Röllin, Parallel Comput. 27 (2001) 883.
[5] http://www.mcs.anl.gov/petsc/.
[6] A. George, J.W. Liu, Computer Solution of Large Sparse Positive Definite Matrices, Prentice Hall, Englewood Cliffs, NJ, 1981.
[7] Y. Saad, M.H. Schultz, SIAM J. Sci. Statist. Comput. 7 (1986) 856.
[8] R.W. Freund, N.M. Nachtigal, Numer. Math. 60 (1991) 315.
[9] S.-W. Huang, T. Carrington Jr., J. Chem. Phys. 112 (2000) 8765.
[10] B. Poirier, T. Carrington Jr., J. Chem. Phys. 114 (2001) 9254.
[11] B. Poirier, T. Carrington Jr., J. Chem. Phys. 116 (2002) 1215.
[12] R. Shepard, A.F. Wagner, J.L. Tilson, M. Minkoff, J. Comput. Phys. 172 (2001) 472.
[13] E.R. Davidson, J. Comput. Phys. 17 (1975) 87.
[14] B. Poirier, W.H. Miller, Chem. Phys. Lett. 265 (1997) 77.
[15] B. Poirier, Phys. Rev. A 56 (1997) 120.
[16] B. Poirier, J. Chem. Phys. 108 (1998) 5216.
[17] R.E. Wyatt, Phys. Rev. E 51 (1995) 3643.
[18] W. Bian, B. Poirier, J. Theor. Comput. Chem. 2 (2003) 583.
[19] W. Bian, B. Poirier, J. Chem. Phys. 121 (2004) 4467.
[20] http://www.lcrc.anl.gov/jazz/index.php.
[21] S. Balay et al., in: E. Arge, A.M. Bruaset, H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, Boston, 1997, pp. 163–202.
[22] S. Balay, et al., Technical Report No. ANL-95/11 – Revision 2.1.5, PETSc Users Manual, Argonne National Laboratory (unpublished).
[23] W.H. Miller, J. Chem. Phys. 62 (1975) 1899.
[24] T. Seideman, W.H. Miller, J. Chem. Phys. 96 (1992) 4412.
[25] U. Manthe, W.H. Miller, J. Chem. Phys. 99 (1993) 3411.
[26] U. Manthe, T. Seideman, W.H. Miller, J. Chem. Phys. 101 (1994) 4759.
[27] http://www-unix.mcs.anl.gov/mpi/mpich2/.
[28] http://www.emsl.pnl.gov/docs/global/ga.html.
[29] http://www.nersc.gov/nusers/resources/SP/.