A Parallel, Block, Jacobi-Davidson Implementation for Solving Large Eigenproblems on Coarse Grain Environments

Andreas Stathopoulos
Department of Computer Science
College of William and Mary
Williamsburg, VA, U.S.A

James R. McCombs
Department of Computer Science
College of William and Mary
Williamsburg, VA, U.S.A

Abstract *Iterative methods often provide the only means of solving large eigenvalue problems. Their block variants converge slowly but they are robust especially in the presence of multiplicities. Preconditioning is often used to improve convergence. Yet, for large matrices, the demands posed on the available computing resources are huge.*

Clusters of workstations and SMPs are becoming the main computational tool for many scientific and engineering groups. In these environments, high communication costs suggest coarse grain implementations of iterative methods.

We have combined the block and preconditioning functionalities into a parallel implementation of a block Jacobi-Davidson method. We combine a fine-grain iterative scheme with the coarse grain capability that each processor can precondition different eigenpairs. We outline the design and present some timings and convergence results on a small workstation cluster and on a SUN Enterprise.

Keywords: Eigenvalue, preconditioning, block, parallel, Jacobi-Davidson

1 Introduction

Many scientific and engineering applications require the computation of a few eigenpairs of large, symmetric matrices. Iterative methods are often the only means of solving these large eigenproblems. Traditionally, Lanczos and subspace iteration [4, 5, 1] have been the two methods of choice. Their strengths are complementary since Lanczos is an efficient algorithm, but it frequently misses multiple eigenvalues, while subspace iteration is robust but slow.

Recently, preconditioning, a powerful technique in the iterative solution of linear systems, has been gaining popularity for eigenproblems. Jacobi-Davidson (JD) [2] is an Arnoldi-like method that can use preconditioning to solve approximately the correction equation for a specific eigenvector. Although, JD can often resolve multiplicities if converged eigenpairs are locked, because of its rapid convergence it misses high multiplicities. Evidently, an efficient and robust combination of JD and subspace iteration is needed.

The increasing availability of inexpensive but powerful workstations and small Shared Memory Multiprocessors (SMPs) has facilitated the migration of many scientific and engineering codes to these environments from the more traditional vector supercomputers and MPPs. Usually, this migration also implies modifying the algorithms and their parallel implementations to effectively use the new environment. Traditionally, iterative methods have been implemented on MPPs in a data parallel way [3]. The power of data parallel implementations is their structured simplicity, involving only global reduction operations and matrix vector multiplications. However, communication and synchronization occurs several times per iteration making these implementations less suitable for coarse grain environments. Developing efficient, coarse grain iterative methods has been challenging because of the sequential nature of the iterations.

Block methods are hybrids of subspace iteration and Lanczos-like methods, where the iterative method is applied simultaneously on a block of vectors. Block methods improve robustness but pay a penalty on the speed of convergence. On parallel environments, block methods provide more computation between communications and thus increase the granularity of the computation. The goal is to counteract the convergence penalty by improved parallel speedup.

In this paper, we outline an implementation of a variable block, JD method, that combines both fine and coarse grain parallelism. The variable block allows for improved robustness in those cases that need it, and the combination of granularities allows for flexible use of the code both on parallel environments with many processors and faster interconnects and on small clusters of workstations with typical Ethernet connections. We present some convergence results and timings to demonstrate the effectiveness of the implementation.

2 A parallel Jacobi-Davidson implementation

Consider the standard eigenvalue problem $Ax_i^* = \lambda_i^* x_i^*$, where A is a large symmetric matrix, and a few extreme (λ_i^*, x_i^*) eigenpairs are required. Although the following discussion considers only symmetric matrices, the results and methods can be trivially extended to the nonsymmetric case. Starting from an initial vector, the JD algorithm successively builds a basis for a space from where the approximations of the eigenpairs are obtained, usually, by the Rayleigh-Ritz procedure [2]. The JD expands the basis by adding the approximate solution t_i of the correction equation (1) for some approximate eigenpair (Ritz pair) (λ_i, x_i):

$$(I - x_i x_i^T)(A - \lambda_i I)(I - x_i x_i^T)t_i = r_i. \qquad (1)$$

$r_i = (A - \lambda_i I)x_i$ is the residual of the Ritz pair.

A block version of the JD starts with a block of k initial vectors, and expands the basis by a block of k vectors at a time. These vectors are the approximate solutions of k correction equations, each for a different Ritz pair.

This algorithm is easily parallelizable in a data parallel way. Each processor keeps a subset of the rows for each basis vector and eigenvector, and performs all vector updates locally. The major difference from the sequential code is that inner products require a global reduction operation (summation), and that a parallel matrix vector multiplication and preconditioning operations must be provided. Below, we outline the data parallel JD algorithm:

JD
V starting k trial vectors, and let $W = AV$
While not converged do:
1. $H = V^T W$ (local contributions)
2. Global_Sum(H) over all processors.
3. Solve $Hy_i = \lambda_i y_i,\ i = 1:k$ (all procs)
4. $x_i = Vy_i,\ z_i = Wy_i, i = 1:k$ (local rows)
5. $r_i = z_i - \lambda_i x_i,\ i = 1:k$ (local rows)
6. **Correction equation** Solve for each t_i
7. Orthogonalize $t_i, i = 1:k$. Add in V
8. **Matrix-vector** $W_i = AV_i,\ i = 1:k$

end while

The easiest way to apply the $(I - x_i x_i^T)$ projections in the correction equation (1) is to use an iterative method for linear systems (such as GMRES or BCGSTAB) and perform the projections before and after the matrix vector multiplication. In our implementation we have used BCGSTAB because it is based on a short recurrence, and thus we can take many steps without having to store a large number of vectors. The BCGSTAB is also implemented in a similar, data parallel way. If a preconditioner is available this can be applied directly to the BCGSTAB for solving the correction equation. Finally, note that the block size k and the number of processors can take arbitrary values. However, for faster convergence, it is better to keep k smaller — and often much smaller — than the number of required eigenpairs.

2.1 A coarse grain interface

In the above data parallel implementation we assume that steps 6 and 8 (correction equation and matrix vector multiplication) are performed through user provided, data parallel functions. However, the above JD design is flexible enough to allow various functions in steps 6, and 8, requiring only that the results are distributed over the processors in a data parallel way (by rows).

Consider the situation where every processor has access to the entire matrix A. This is often the case, not only when A is very sparse, but especially in applications that do not store the matrix, providing instead a function for computing the matrix vector product. In these applications, matrix storage is not an issue. A typical example (and our initial motivation for this implementation) comes from the area of materials science [6]. When discretizing the Schrödinger operator on a planewave basis, the matrix can be represented as two diagonals. The problem is that one of these diagonals is in real space and the other in Fourier space. To compute the action of the matrix on a vector, we transform the vector from Fourier to real space, and transform the result back to Fourier space. Thus the matrix vector multiplication consists of two vector updates and two FFTs.

If a processor has access to A, and also stores all the rows of a vector, the matrix vector product can be performed locally (sequentially). The parallelism comes from having each processor gather all the rows of one of the block vectors and apply the matrix vector product independently from other processors. Assuming for simplicity that the block size is equal to the number of processors, the following is an example of a coarse grain step 8.

8. **Coarse grain matrix-vector**
   Perform all-to-all communication:
      send local piece of V_i to proc i
      receive a piece for V_{myid} from proc i
   Perform $W_{myid} = AV_{myid}$ locally
   Perform all-to-all communication:
      send the i-th piece of W_{myid} to proc i
      receive a piece for W_i from proc i

Although the matrix vector products are performed in parallel, the expensive all-to-all personalized communication limits the scalability for general sparse matrices. However, the power of this interface is in the coarse grain solution of the correction equations for the vectors in the block. If each processor gathers all the rows of a vector, it can apply any number of preconditioned BCGSTAB steps completely independently from other processors. Thus, all the correction equations for the block are solved in parallel. The following is an example of this coarse grain interface.

6. **Coarse grain correction equation**
   Perform all-to-all communication:
      send local pieces of x_i, r_i to proc i
      receive a piece for x_{myid}, r_{myid} from proc i
   Apply m steps of (preconditioned) BCGSTAB
      on eq.(1) with the gathered r_{myid}
   Perform all-to-all communication:
      send the i-th piece of t_{myid} to proc i
      receive a piece for t_i from proc i

In this method, larger number of BCGSTAB steps (m) means more parallel work between communications. Therefore, by increasing m we can improve arbitrarily the parallel speedup of the method. Typically, we choose a small m for easy problems (requiring few iterations), and a larger m for hard problems. In addition, we gradually increase m in later JD iterations.

As mentioned earlier, the communication requirements for the all-to-all phase could be excessive for general sparse matrices, where matrix vector product requires communication only of a small subset of the vector V. However, in cases such as the one from materials science the matrix is not sparse, and the product is performed through the use of FFTs or a similar complicated function. In this case, the communication requirements of the FFTs are comparable to the all-to-all of our methods.

2.2 Other implementation features

For portability and efficiency, we have implemented the above code in Gnu Fortran 77, using MPI message passing. In step 7, we use Gram-Schmidt with re-orthogonalization to assure both orthogonality and efficient parallelization. Small eigenproblems in step 3 are solved by LAPACK calls. We also adopt a locking and window technique for finding a large number of eigenpairs by utilizing only small basis sizes. Finally, we apply thick restarting [7] when the size of the basis V reaches a user specified upper limit.

The global summations in the JD driver are delayed as much as possible, and are applied on blocks of partial results. Thus, the code has 1 synchronization point at step 2, and 2 additional ones in orthonormalization (step 7). An additional synchronization point occurs whenever the need for reorthogonalization emerges. From the above, the code is well tuned as a fine grained method, and if data parallel matrix vector product and preconditioner are provided, the code is scalable to a large number of processors on MPPs.

The flexibility of the code stems from the coarse grain interfaces discussed earlier. These are particularly useful in computational environments with high latencies and low bandwidths, such as clusters of workstations and PCs. The block size in our implementation can vary from 1 (fine grain JD) to the maximum basis size (preconditioned subspace iteration). Therefore, the block size can be considered a tuning knob between different methods. Finally, the code handles any combination of number of processors and block size. The implementation involves the dynamic creation of processor subgroups and its details are not discussed in this paper. The experiments in the next section reflect the simple case where the number of processors is equal to the block size.

3 Timing experiments

We have tested the above code with various choices of block size k, number of BCGSTAB iterations m, with or without preconditioning, and on two different parallel environments. We have also compared against PARPACK, an efficient data parallel implementation of the popular ARPACK package. Both PARPACK and JD use a maximum basis size of 25, and five smallest or largest eigenpairs are required. The residual tolerance is set to $\|A\|_F 10^{-14}$. To facilitate the table illustration of the comparisons we adopt the following notation:

- p — Number of processors.
- PAR — The PARPACK method.
- JDb(m) — The block JD, with block size k =p. Each JD iteration performs m steps of unpreconditioned BCGSTAB on each vector in the block.
- JDb(m)-i — As with JDb(m), only the BCGSTAB steps are preconditioned with ILUT.
- JDb(m)-d — As with JDb(m)-i, only diagonal preconditioning is used instead.
- JD(0) — Fine grain JD, block size $k = 1$, and no correction equation (Arnoldi).

We have run experiments on two parallel machines. The first is a SUN Enterprise 450 server with 4 processors. The processors are UltraSparc II 300 MHz, with 2 MB level 2 cache each. The total available memory is 512 MB, and is accessible through a crossbar switch. The second, is a cluster of three PCs, connected through Ethernet, each with a Pentium II, 400 MHz processor, 512 KB level 2 cache, and 360 MB of memory. The sizes of these machines are rather small, but typical of the computational resources available to many small scientific and engineering groups.

We used two symmetric test matrices in Harwell-Boeing format, which are available from MatrixMarket[1] We favor these over matrices from materials science because it is easier for other researchers to confirm and juxtapose our results. The first matrix is NASASRB of dimension 54870, and with 2677324 non zero elements. While the largest part of its spectrum is easily obtained, the lower part is extremely clustered and difficult to converge to. The second matrix is the BCSSTK16 of dimension 4884, and with 290378 non zero elements. The interesting characteristic is that its lowest

[1] URL: `http//math.nist.gov/MatrixMarket/`

eigenvalue (1) has a very high numerical multiplicity (more than 40).

3.1 Testing scalability

Demonstrating parallel scalability of the implementation is complicated because different numbers of processors have different block sizes, and thus different convergence behaviors. First, we simply demonstrate the scalability of the JD driver, by ignoring convergence and stopping the codes after a certain number of matrix vector products. To test solely the scalability of the driver, we use two diagonal matrices of sizes 5000 and 50000 that incur no communication in the matrix vector function. All tests are versus the PARPACK.

The speedups in Table 1 show that the JD is at least as scalable as PARPACK, despite the more complicated step. The `MPI_alltoall` in the matrix vector product of the NASASRB proves communication intensive for both methods. In fact, neither of the fine grain methods can exploit parallelism on the PC cluster. In contrast, the coarse grain JD can achieve arbitrarily high speedup by increasing the independent number of BCGSTAB steps.

3.2 Timings for an easy problem

Because the number of matrix vector products typically increases with the block size, the scalability observed in the previous section does not always translate into speedup. In Table 2 we show results from converging to the five largest eigenpairs of NASASRB on the SUN SMP. Beyond scalability, we see that PARPACK and JD(0) have similar convergence characteristics. Although identical in theory, JD(0) typically results in fewer iterations while PARPACK in lower times.

The coarse grain algorithm turns out to be an overkill for this problem. As expected for JDb(10), the number of iterations decreases but the number of matrix vector products increases both with block size and BCGSTAB steps. Therefore, four processors perform more work than one processor and the observed parallel efficiency is less than 50%. This problem is too easy to require a sophisticated block, inner-outer method.

3.3 Timings for a difficult problem

The lowest eigenvalues of the NASASRB have relative gap ratios that are on the order of 10^{-10}, making their computation by Krylov methods a hard problem. After 350000 matrix vector products, PARPACK was far from convergence. Applying shift and invert on large, sparse matrices is usually not possible or it involves excessive memory and time requirements. On the other hand, powerful preconditioners such as incomplete factorization, are often easily computed. This is a significant advantage of the JD variants. For NASASRB, the ILUT$(20, 0)$, which is the SPARSKIT incomplete factorization with 0 threshold, is computed in merely 30 seconds (both on the SUN and the PCs). For stability, we computed the ILUT of the shifted matrix $A + 10^6 I$. The results in Table 3 show that the coarse grain algorithm using ILUT-preconditioned BCGSTAB converges to the five lowest eigenpairs in less than one hour, both on the SUN and the PCs. As before, the number of matrix vector multiplications increases with the block size, but so does the parallel speedup.

It is interesting to note that for difficult problems, a larger number of BCGSTAB steps improves convergence. This is in contrast with the results in Table 2. Dynamically deciding the number of BCGSTAB steps according to the difficulty of the problem would be a desirable feature of robust JD implementations. Finally, it is worth noting, that the coarse granularity of the method masks completely the large Ethernet latencies on PCs, providing better timings and speedups than the SUN SMP.

3.4 Resolving high multiplicities

Besides better computational and parallel efficiency, the strength of subspace iteration and block methods lies in their ability to find eigenpairs with high multiplicities. We confirm this

SUN speedups:

	NASASRB			Diag(5000)		Diag(50000)	
p	PAR	JDb(150)	JD(0)	PAR	JD(0)	PAR	JD(0)
2	1.94	1.97	1.90	1.60	1.75	2.01	2.01
3	2.65	2.80	2.64	1.99	2.30	3.17	3.09
4	3.34	3.42	3.55	2.03	2.55	4.26	4.26

PC speedups:

	NASASRB			Diag(5000)		Diag(50000)	
p	PAR	JDb(150)	JD(0)	PAR	JD(0)	PAR	JD(0)
2	0.91	1.84	1.06	1.79	1.74	2.09	2.16
3	0.61	2.36	0.71	2.01	1.99	3.09	3.26

Table 1: SUN Enterprise and PC cluster speedups of 2, 3, and 4 processors over 1 processor, for three different matrices.

	PAR			JD(0)			JDb(10)-d		
p	Mvec	Time	Sp	Mvec	Time	Sp	Mvec	Time	Sp
1	215	153.45	1	162	218.78	1	724	246.56	1
2	215	79.21	1.94	161	116.46	1.90	824	160.95	1.53
3	215	57.81	2.65	162	82.74	2.64	880	126.79	1.95
4	215	45.98	3.34	164	61.52	3.55	1070	131.11	1.88

Table 2: Matrix vector products (Mvec), time, and speedup (Sp) for five largest eigenpairs of the NASASRB on the SUN Enterprise.

strength in our variable block JD implementation by running an experiment on BCSSTK16 from Harwell-Boeing. The numerical multiplicity of the lowest eigenvalue is more than 40. As Table 4 shows, typical Krylov methods such as PARPACK fail to find any multiplicities. Instead, the five smallest distinct eigenpairs are computed. However, the JD variants are able to locate 3 multiplicities even without using a block method (block size = 1 = # processors). This is achieved by requiring a low convergence tolerance, and by explicitly locking converged eigenpairs. Even when the tolerance is relaxed to $\|A\|_F 10^{-8}$, the JDb(10) still computes two multiple eigenvalues. By increasing the number of processors (and thus the block size) a larger number of multiplicities can be obtained. This suggests a robust behavior of the JDb methods without necessarily paying the price of a larger block size.

4 Conclusions

We have outlined an implementation of a robust and efficient, variable block, Jacobi-Davidson method that can be used in a variety of computational environments. The robustness is the outcome of several state-of-the-art techniques for blocking, orthogonalization, restarting, and locking. The flexibility stems from interfacing with various matrix vector multiplication and preconditioning functions. Sequential implementations of these functions allow for an efficient use on a uniprocessor, while data parallel implementations facilitate runs on massively parallel computers. The implementation is at least as scalable as other data parallel codes such as PARPACK.

In addition, we have provided a novel interface for the code by applying the correction equation for each vector in the block in-

p	JDb(150)-i on SUN			JDb(5)-i on SUN			JDb(150)-i on PCs		
	Mvec	Time	Sp	Mvec	Time	Sp	Mvec	Time	Sp
1	10815	6052	1	12077	8446	1	9989	6108	1
2	12581	3784	1.60	27425	10065	0.84	11777	3659	1.67
3	13570	3038	1.99	30814	8349	1.01	13804	3013	2.03
4	19517	3525	1.72	57233	12429	0.68	-	-	-

Table 3: Matrix vector products (Mvec), time, and speedup (Sp) for five lowest eigenpairs of the NASASRB. The BCGSTAB is preconditioned by ILUT(20,0).

	Tol: $\|A\|_F 10^{-14}$									Tol: $\|A\|_F 10^{-8}$		
	PAR			JDb(100)			JDb(10)			JDb(10)		
p	Mvec	Time	Mult	Mvec	Time	Mult	Mvec	Time	multpc	Mvec	Time	Mult
1	115	93.6	1	2749	76.2	3	1645	54.9	3	1209	39.2	2
2	115	52.4	1	2621	44.9	4	2155	37.9	4	1859	32.4	4
4	115	36.5	1	5029	50.8	5	4141	41.4	5	2663	26.3	5

Table 4: Matrix vector products (Mvec), time, and number of multiplicities found (Mult) for the BCSSTK16 matrix on the SUN Enterprise. Results from two residual tolerences are reported.

dependently. Although this may result in an increase of the number of matrix vector products, it dramatically improves the coarse granularity of the algorithm. The resulting code can achieve high parallel efficiencies even on coarse grain environments with high latencies, such as clusters of workstations. Finally, this variable block code can be used to find eigenpairs with high multiplicities.

References

[1] J. Cullum and W.E. Donath. A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, symmetric matrices. In *Proc. 1974 IEEE Conference on Decision and Control*, pages 505–509, 1974.

[2] D. R. Fokkema, G. L. G. Sleijpen, and H. A. van der Vorst. Jacobi-Davidson style QR and QZ algorithms for the partial reduction of matrix pencils. Technical Report 941, Department of Mathematics, University of Utrecht, 1996. To appear in SIAM J. Sci. Comput.

[3] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* SIAM, Philadelphia, PA, 1998.

[4] D. P. O'Leary. Parallel implementation of the block conjugate gradient algorithm. *Parallel Computing*, 5:127–139, 1987.

[5] B. N. Parlett. *The Symmetric Eigenvalue Problem.* SIAM, Philadelphia, PA, 1998.

[6] A. Stathopoulos, S. Öğüt, Y. Saad, J. R. Chelikowsky, and H. Kim. Parallel methods and tools for predicting material properties. *IEEE Computational Science & Engineering, to appear*, 1999.

[7] A. Stathopoulos, Y. Saad, and K. Wu. Dynamic thick restarting of the Davidson, and the implicitly restarted Arnoldi methods. *SIAM J. Sci. Comput.*, 19:227–245, 1998.