PARPACK: An Efficient Portable Large Scale Eigenvalue Package for Distributed Memory Parallel Architectures

K. J. Maschhoff
D. C. Sorensen

CRPC-TR96659
October 1996

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

revised November 1996

P_ARPACK: An Efficient Portable Large Scale Eigenvalue Package for Distributed Memory Parallel Architectures

K. J. Maschhoff and D. C. Sorensen

Rice University

Abstract. P_ARPACK is a parallel version of the ARPACK software. ARPACK is a package of Fortran 77 subroutines which implement the Implicitly Restarted Arnoldi Method used for solving large sparse eigenvalue problems. A parallel implementation of ARPACK is presented which is portable across a wide range of distributed memory platforms and requires minimal changes to the serial code. The communication layers used for message passing are the Basic Linear Algebra Communication Subprograms (BLACS) developed for the ScaLAPACK project and Message Passing Interface(MPI).

1 Introduction

ARPACK is a collection of Fortran77 subroutines designed to solve large scale eigenvalue problems. ARPACK stands for ARnoldi PACKage. ARPACK software is capable of solving large scale non Hermitian (standard and generalized) eigenvalue problems from a wide range of application areas. The software is designed to compute a few, say k, eigenvalues with user specified features such as those of largest real part or largest magnitude using only $n \cdot \mathcal{O}(k) + \mathcal{O}(k^2)$ storage. A set of Schur basis vectors for the desired k dimensional eigen-space is computed which is numerically orthogonal to working precision. Eigenvectors are also available upon request. Parallel ARPACK (P_ARPACK) is provided as an extension to the current ARPACK library and is targeted for distributed memory message passing systems. The message passing layers currently supported are BLACS and MPI.

The Arnoldi process is a technique for approximating a few eigenvalues and corresponding eigenvectors of a general $n \times n$ matrix. It is most appropriate for large structured matrices $\mathbf{A}$ where structured means that a matrix-vector product $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}$ requires $\mathcal{O}(n)$ rather than the usual $\mathcal{O}(n^2)$ floating point operations (Flops). This software is based upon an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method (IRAM). When the matrix $\mathbf{A}$ is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method (IRLM). These variants may be viewed as a synthesis of the Arnoldi/Lanczos process with the Implicitly Shifted QR technique that is suitable for large scale problems. For many standard problems, a matrix factorization is not required. Only the action of the matrix on a vector is needed.

The important features of ARPACK and P_ARPACK are:

- A reverse communication interface.
- Ability to return k eigenvalues which satisfy a user specified criterion such as largest real part, largest absolute value, largest algebraic value (symmetric case), etc. For many standard problems, the action of the matrix on a vector $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}$ is all that is needed.
- A fixed pre-determined storage requirement suffices throughout the computation. Usually this is $n \cdot \mathcal{O}(k) + \mathcal{O}(k^2)$ where k is the number of eigenvalues to be computed and n is the order of the matrix. No auxiliary storage or interaction with such devices is required during the course of the computation.
- Sample driver routines are included that may be used as templates to implement various spectral transformations to enhance convergence and to solve the generalized eigenvalue problem.
- Special consideration is given to the generalized problem $\mathbf{A}\mathbf{x} = \mathbf{M}\mathbf{x}\lambda$ for singular or ill-conditioned symmetric positive semi-definite $\mathbf{M}$.
- Eigenvectors and/or Schur vectors may be computed on request. A Schur basis of dimension k is always computed. The Schur basis consists of vectors which are numerically orthogonal to working accuracy. Computed eigenvectors of symmetric matrices are also numerically orthogonal.
- The numerical accuracy of the computed eigenvalues and vectors is user specified. Residual tolerances may be set to the level of working precision. At working precision, the accuracy of the computed eigenvalues and vectors is consistent with the accuracy expected of a dense method such as the implicitly shifted QR iteration.
- Multiple eigenvalues offer no theoretical or computational difficulty other than additional matrix-vector products required to expose the multiple instances. This is made possible through the implementation of deflation techniques similar to those employed to make the implicitly shifted QR algorithm robust and practical. Since a block method is not required, the user does not need to "guess" the correct block size that would be needed to capture multiple eigenvalues.

2 Parallelizing ARPACK

The parallelization paradigm found to be most effective for ARPACK on distributed memory machines was to provide the user with a Single Program Multiple Data (SPMD) template. The reverse communication interface is one of the most important aspects in the design of ARPACK and this feature lends itself to a simplified SPMD parallelization strategy. This approach was used for previous parallel implementations of ARPACK [2] and is simple for the the user to implement. The reverse communication interface feature of ARPACK allows the P_ARPACK codes to be parallelized internally without imposing a fixed parallel decomposition on the matrix or the user supplied matrix-vector product.

Memory and communication management for the matrix-vector product can be optimized independent of P_ARPACK . This feature enables the use of various matrix storage formats as well as calculation of the matrix elements on the fly.

The calling sequence to ARPACK remains unchanged except for the addition of the BLACS context (or MPI communicator). Inclusion of the context (or communicator) is necessary for global communication as well as managing I/O. The addition of the context is new to this implementation and reflects the improvements and standardizations being made in message passing [9, 7].

2.1 Data Distribution of the Arnoldi Factorization

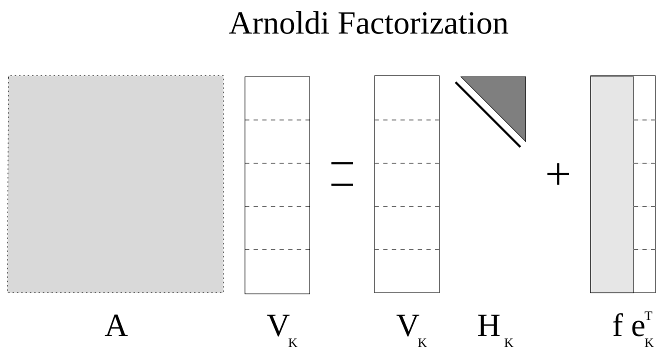
The numerically stable generation of the Arnoldi factorization

$$\mathbf{A}\mathbf{V}_k = \mathbf{V}_k\mathbf{H}_k + \mathbf{f}_k\mathbf{e}_k^T$$

where

$\mathbf{A}$, $n \times n$ matrix
$\mathbf{H}_k$, $k \times k$ - projected matrix (Upper Hessenberg)
$\mathbf{V}_k$, $n \times k$ matrix, $k \ll n$ - Set of Arnoldi vectors
$\mathbf{f}_k$, residual vector, length n, $\mathbf{V}_k^T\mathbf{f}_k = 0$

coupled with an implicit restarting mechanism [1] is the basis of the ARPACK codes. The simple parallelization scheme used for P_ARPACK is as follows.



- $\mathbf{H}_k$ replicated on every processor
- $\mathbf{V}_k$ is distributed across a 1-D processor grid. (Blocked by rows)
- $\mathbf{f}_k$ and workspace distributed accordingly

The SPMD code looks essentially like the serial code except that the local block of the set of Arnoldi vectors, $\mathbf{V}_{loc}$, is passed in place of $\mathbf{V}$, and n_{loc}, the dimension of the local block, is passed instead of n.

With this approach there are only two communication points within the construction of the Arnoldi factorization inside P_ARPACK: computation of the 2-norm of the distributed vector f_k and the orthogonalization of f_k to V_k using Classical Gram Schmidt with DGKS correction [5]. Additional communication will typically occur in the user supplied matrix-vector product operation as well. Ideally, this product will only require nearest neighbor communication among the processes. Typically the blocking of $\mathbf{V}$ is commensurate with the parallel decomposition of the matrix $\mathbf{A}$. The user is free, however, to select an appropriate blocking of $\mathbf{V}$ such that an optimal balance between the parallel performance of P_ARPACK and the user supplied matrix-vector product is achieved.

The SPMD parallel code looks very similar to that of the serial code. Assuming a parallel version of the subroutine matvec, an example of the application of the distributed interface is illustrated as the follows:

```
 10  continue
     call psnaupd (comm, ido, bmat, nloc, which, ...,
    *                      Vloc , ... lworkl, info)
     if (ido .eq. newprod) then
        call matvec ('A', nloc, workd(ipntr(1)), workd(ipntr(2)))
     else
        return
     endif
     go to 10
```

Where, `nloc` is the number of rows in the block `Vloc` of V that has been assigned to this node process.

Typically, the blocking of V is commensurate with the parallel decomposition of the matrix A as well as with the configuration of the distributed memory and interconnection network. Logically, the V matrix be partitioned by blocks

$$\mathbf{V}^T = (\mathbf{V}^{(1)^T}, \mathbf{V}^{(2)^T}, ..., \mathbf{V}^{(nproc)^T})$$

with one block per processor and with $\mathbf{H}$ replicated on each processor. The explicit steps of the process responsible for the j block are shown in Table 1.

Note that the function *gnorm* at Step 1 is meant to represent the global reduction operation of computing the norm of the distributed vector $\mathbf{f}_k$ from the norms of the local segments $\mathbf{f}_k^{(j)}$ and the function *gsum* at Step 3 is meant to represent the global sum of the local vectors $\mathbf{h}^{(j)}$ so that the quantity $\mathbf{h} = \sum_{j=1}^{nproc} \mathbf{h}^{(j)}$ is available to each process on completion. These are the only two communication points within this algorithm. The remainder is perfectly parallel. Additional communication will typically occur at Step 2. Here the operation $(\mathbf{A}loc)v$ is meant to indicate that the user supplied matrix-vector product is able to compute the local segment of the matrix-vector product $\mathbf{Av}$ that is consistent with the partition of $\mathbf{V}$. Ideally, this would only involve nearest neighbor communication among the processes.

(1) $\beta_k \leftarrow gnorm(\|f_k^{(*)}\|)$; $v_{k+1}^{(j)} \leftarrow f_k^{(j)} \cdot \frac{1}{\beta_k}$;

(2) $w^{(j)} \leftarrow (Aloc)v_{k+1}^{(j)}$;

(3) $\begin{pmatrix} h \\ \alpha \end{pmatrix}^{(j)} \leftarrow \begin{pmatrix} V_k^{(j)T} \\ v_{k+1}^{(j)T} \end{pmatrix} w^{(j)}$; $\begin{pmatrix} h \\ \alpha \end{pmatrix} \leftarrow gsum \left[\begin{pmatrix} h \\ \alpha \end{pmatrix}^{(*)} \right]$

(4) $f_{k+1}^{(j)} \leftarrow w^{(j)} - (V_k, v_{k+1})^{(j)} \begin{pmatrix} h \\ \alpha \end{pmatrix}$;

(5) $H_{k+1} \leftarrow \begin{pmatrix} H_k & h \\ \beta_k \; e_k^T & \leftarrow \end{pmatrix}$;

(6) $V_{k+1}^{(j)} \leftarrow (V_k, v_{k+1})^{(j)}$;

Table 1. The explicit steps of the process responsible for the j block.

Since **H** is replicated on each processor, the parallelization of the implicit restart mechanism described in [1, 2] remains untouched. The only difference is that the local block $\mathbf{V}^{(j)}$ is in place of the full matrix $\mathbf{V}$. All operations on the matrix **H** are replicated on each processor. Thus there is no communication overhead but there is a "serial bottleneck" here due to the redundant work. If k is small relative to n this bottleneck is insignificant. However, it becomes a very important latency issue as k grows and will prevent scalability if k grows with n as the problem size increases.

The main benefit of this approach is that the changes to the serial version of ARPACK were very minimal. Since the change of dimension from matrix order n to its local distributed blocksize `nloc` is invoked through the calling sequence of the subroutine `psnaupd`, there is no fundamental algorithmic change to the code. Only eight routines were affected in a minimal way. These routines either required a change in norm calculation to accomodate distributed vectors (Step 1), modification of the distributed dense matrix-vector product (Step 4), or inclusion of the context or communicator for I/O (debugging/tracing). More specifically, the commands are changed from

```
   rnorm = sdot (n, resid, 1, workd, 1)
   rnorm = sqrt(abs(rnorm))
```

to

```
   rnorm = sdot (n, resid, 1, workd, 1)
   call sgsum2d(comm,'All',' ',1, 1, rnorm, 1, -1, -1 )
   rnorm = sqrt(abs(rnorm))
```

where `sgsum2d` is the BLACS global sum operator. The MPI implementation uses the MPI_ALLREDUCE global operator. Similarly, the computation of the matrix-vector product operation $h \leftarrow V^T w$ requires a change from

```
   call sgemv ('T', n, j, one, v, ldv, workd(ipj), 1,
  *           zero, h(1,j), 1)
```

to

```
   call sgemv ('T', n, j, one, v, ldv, workd(ipj), 1,
  *           zero, h(1,j), 1)
   call sgsum2d( comm, 'All', ' ', j, 1, h(1,j), j,
  *             -1, -1 )
```

Another strategy which was tested was to use Parallel BLAS (PBLAS) [8] software developed for the ScaLAPACK project to achieve parallelization. The function of the PBLAS is to simplify the parallelization of serial codes implemented on top of the BLAS. The ARPACK package is very well suited for testing this method of parallelization since most of the vector and matrix operations are accomplished via BLAS and LAPACK routines.

Unfortunately this approach required additional parameters to be added to the calling sequence (the distributed matrix descriptors) as well as redefining the workspace data structure. Although there is no significant degradation in performance, the additional code modifications, along with the data decomposition requirements, make this approach less favorable. As our parallelization is only across a one dimensional grid, the functionality provided by the PBLAS was more sophisticated than we required. The current implementation of the PBLAS (ScaLAPACK version 1.1) assumes the matrix operands to be distributed in a block-cyclic decomposition scheme.

2.2 Message Passing

One objective for the development and maintenance of a parallel version of the ARPACK [3] package was to construct a parallelization strategy whose implementation required as few changes as possible to the current serial version. The basis for this requirement was not only to maintain a level of numerical and algorithmic consistency between the parallel and serial implementations, but also to investigate the possibility of maintaining the parallel and serial libraries as a single entity.

On many shared memory MIMD architectures, a level of parallelization can be accomplished via compiler options alone without requiring any modifications to the source code. This is rather ideal for the software developer. For example, on the SGI Power Challenge architecture the MIPSpro F77 compiler uses a POWER FORTRAN Accelerator (PFA) preprocessor to automatically uncover the parallelism in the source code. PFA is an optimizing Fortran preprocessor that discovers parallelism in Fortran code and converts those programs to parallel code. A brief discussion of implementation details for ARPACK using PFA preprocessing may be found in [6]. The effectiveness of this preprocessing step is still dependent on how suitable the source code is for parallelization. Since most of the vector and matrix operations for ARPACK are accomplished via BLAS and LAPACK routines, access to efficient parallel versions of these libraries alone will provide a reasonable level of parallelization.

Unfortunately, for distributed memory architectures the software developer is required to do more work. For distributed memory implementations, message passing between processes must be explicitly addressed within the source code and numerical computations must take into account the distribution of data. In addition, for the parallel code to be portable, the communication interface used for message passing must be supported on a wide range of parallel machines and platforms. For /small P_ARPACK, this portability is achieved via the Basic Linear Algebra Communication Subprograms (BLACS) [7] developed for the ScaLAPACK project and Message Passing Interface (MPI) [9].

3 Parallel Performance

To illustrate the potential scalability of Parallel ARPACK on distributed memory architectures some example problems have been run on the Maui HPCC SP2. The results shown in Table 1 attempt to illustrate the potential internal performance of the of the P_ARPACK routines independent of the users implementation of the matrix vector product.

In order to isolate the performance of the ARPACK routines from the performance of the user's matrix-vector product and also to eliminate the effects of a changing problem characteristic as the problem size increases, a test was comprised of replicating the same matrix repeatedly to obtain a block diagonal matrix. This completely contrived situation allows the workload to increase linearly with the number of processors. Since each diagonal block of the matrix is identical, the algorithm should behave as if *nproc* identical problems are being solved simultaneously (provided an appropriate starting vector is used). For this example we use a starting vector of all "1's". The only obstacles which prevent ideal speedup are the communication costs involved in the global operations and the "serial bottleneck" associated with the replicated operations on the projected matrix H. If neither of these were present then one would expect the execution time to remain constant as the problem size and the number of processors increase.

The matrix used for testing is a diagonal matrix of dimension $100,000$ with uniform random elements between 0 and 1 with four of the diagonal elements separated from the rest of the spectrum by adding an additional 1.01 to these elements. The problem size is then increased linearly with the number of processors by adjoining an additional diagonal block for each additional processor. For these timings we used the non-symmetric P_ARPACK code pdnaupd with the following parameter selections: mode is set to 1, number of Ritz values requested is 4, portion of the spectrum is "LM", and the maximum number of columns of V is 20.

Number of Nodes	Problem Size	Total Time (s)	Efficiency
1	100,000 * 1	40.53	
4	100,000 * 4	40.97	0.98
8	100,000 * 8	42.48	0.95
12	100,000 * 12	42.53	0.95
16	100,000 * 16	42.13	0.96
32	100,000 * 32	46.59	0.87
64	100,000 * 64	54.47	0.74
128	100,000 * 128	57.69	0.70

Table 2. Internal Scalability of P_ARPACK

4 Availability

The codes are available by anonymous ftp from

```
ftp.caam.rice.edu
```

The ARPACK package is in

```
pub/software/ARPACK/arpack96.tar.gz
```

Parallel ARPACK (P_ARPACK) is in

```
pub/software/ARPACK/parpack96.tar.gz
```

Follow the instructions in the README files. ARPACK and P_ARPACK software is also available on Netlib in the directory `scalapack`.

5 Summary

We have presented a parallel implementation of the ARPACK library which is portable across a wide range of distributed memory platforms. The portability of P_ARPACK is achieved by utilization of the BLACS and MPI. We have been quite satisfied with how little effort it takes to port P_ARPACK to a wide variety of parallel platforms. So far we have tested P_ARPACK on a SGI Power Challenge cluster using PVM-BLACS and MPI, on a CRAY T3D using Cray's implementation of the BLACS, on an IBM SP2 using MPL-BLACS and MPI, on a Intel paragon using NX-BLACS and MPI, and on a network of Sun stations using MPI and MPI-BLACS.

6 Research Funding of ARPACK

Financial support for this work was provided in part by the National Science Foundation cooperative agreement CCR-912008, and by the ARPA contract number DAAL03-91-C-0047 (administered by the U.S. Army Research Office).

References

1. D. C. Sorensen, *Implicit application of polynomial filters in a k-step Arnoldi method,* SIAM Journal on Matrix Analysis and Applications, 13(1):357-385, January 1992.
2. D. C. Sorensen, *Implicitly-Restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations,* (invited paper), in Parallel Numerical Algorithms: Proceedings of an ICASE/LaRC Workshop, May 23-25, 1994, Hampton, VA, D. E. Keyes, A. Sameh, and V. Venkatakrishnan, eds., Kluwer, 1995 (to appear).
3. R.B. Lehoucq, D.C. Sorensen, P.A. Vu, and C. Yang, *ARPACK: Fortran subroutines for solving large scale eigenvalue problems,* Release 2.1
4. R. B. Lehoucq and D.C. Sorensen *Deflation Techniques for an Implicitly Re-started Arnoldi Iteration* , To appear in SIAM Journal of Matrix Analysis
5. J. Daniel, W.B. Gragg, L. Kaufman, and G.W. Stewart *Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization* , Mathematics of Computation, 30:772-795, 1976
6. M.P. Debicki, P. Jedrzejewski, J. Mielewski, P. Przybyszewski, and M. Mrozowski *Application of the Arnoldi Method to the Solution of Electromagnetic Eigenproblems on the Multiprocessor Power Challenge Architecture,* Technical Report Number 19/95, Department of Electronics, Technical University of Gdansk, Poland.
7. J. J. Dongarra and R. C. Whaley *LAPACK Working Note 94, A User's Guide to the BLACS v1.0,* , June 7, 1995
8. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley *LAPACK Working Note 100, A Proposal for a Set of Parallel Basic Linear Algebra Subprograms* , May 1995
9. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard* , International Journal of Supercomputer Applications and High Performance Computing, 8(3/4), 1994

This article was processed using the LaTeX macro package with LLNCS style