IMPLEMENTATION OF THE PARALLEL ARNOLDI METHOD IN THE IBM SP2 DISTRIBUTED MEMORY SYSTEM

MICHAŁ REWIEŃSKI

The Technical University of Gdańsk
Department of Electronics, Telecommunications and Computer Science

Abstract: The following article discusses the Implicitly Restarted Arnoldi Method used for solving large sparse eigenvalue problems. It presents the parallel implementation of this algorithm for a distributed memory architecture developed by Maschhoff and Sorensen ([1]) and included in the P_ARPACK library. The article gives results of performance tests of the P_ARPACK subroutines in the IBM Scalable POWER2 (SP2) parallel system installation at the Academic Computer Centre TASK in Gdańsk and describes some technical problems concerning use of message-passing libraries (particularly the Message Passing Interface (MPI)), as well as communication subsystems available in IBM SP2 with the discussed software package.

1. Introduction

The Arnoldi method belongs to a class of iterative algorithms capable of computing a few eigenvalues and eigenvectors of non-Hermitian matrices. It provides an efficient tool for solving problems from different application areas that are reducible to finite-dimensional operator equations. This method is most suited for finding eigenvalues of large structured matrices where the matrix-vector product operation requires $O(n)$ rather than $O(n^2)$ floating point operations.

The ARPACK software package developed at Rice University (cf. [2] and [3]) provides an implementation of a version of Arnoldi algorithm called the Implicitly Restarted Arnoldi Method (IRAM) which for symmetric problems reduces to the Implicitly Restarted Lanczos Method (IRLM). One of the most important features of this package of Fortran77 subroutines is that it allows the user to compute a few eigenvalues from a specified part of the operator spectrum. Another advantage of this implementation is a pre-determined memory complexity which equals $nO(k)+O(k^2)$, where k is the number of required eigenvalues and n is the order of the input matrix operator. The Arnoldi method, as most of the iterative processes, does not require any explicit form of the input operator matrix to be given. Instead all the information on the considered operator is passed via the matrix-vector product. This characteristic of the algorithm has been efficiently used in the ARPACK software by introducing

the *reverse communication* interface. On one hand this interface enables the subroutines that perform the Arnoldi algorithm iteration to be independent of the input matrix storage format and on the other hand it makes the user of ARPACK free to choose the most appropriate method for computing the matrix-vector product. This versatile interface certainly broadens the area for applications of Arnoldi algorithm and may often contribute to a considerable improvement of performance of solvers in many specific scientific and engineering problems.

Although the original implementation of the Arnoldi method consisted of sequential codes, it proved to be useful and efficient also in multiprocessor parallel systems. In the report by Dębicki et al. [4] parallelization strategies as well as results of performance tests of ARPACK software have been given for the SGI Power Challenge shared memory system. Recently the subroutines of the Arnoldi package have been parallelized for use in multiprocessor distributed memory systems which were found to provide the most suitable working environment for concurrent execution of this software. A new library called P_ARPACK (Parallel Arnoldi Package) has been developed enabling the use of a Single Program Multiple Data (SPMD) style which is regarded the most effective and transparent in the message-passing programming. Once again the reverse communication interface to the P_ARPACK subroutines enables the user to choose a convenient parallelization strategy for the matrix-vector product operation. Currently the communication libraries which may be used with P_ARPACK are the Basic Linear Algebra Communication Subprograms (BLACS) and Message Passing Interface (MPI). This is a great advantage of this software as both libraries are portable across a wide range of computer systems. Moreover, MPI emerges as a standard tool in the message-passing programming (cf. [5]).

One of the modern hardware environments very well suited for testing of algorithms that exploit the message-passing programming paradigm is the IBM Scalable POWER2 (SP2) parallel system. This computer architecture is a very typical implementation of a distributed memory machine and provides a set of software tools, including e.g. MPI library, needed to handle the execution of message-passing based parallel programs. Therefore it may be applied for the performance tests of various parallel numerical algorithms or entire libraries that make use of the data distribution and message-passing as the primary means of parallelization.

Such tests have been made in the case of the parallel Arnoldi package using a 16-processor IBM SP2 machine installation at the Academic Computer Centre TASK in Gdańsk in order to measure the performance of the P_ARPACK library, as well as to discuss some issues concerning running parallel message-passing programs in the mentioned system.

2. The Arnoldi method

The Arnoldi method belongs to a class of iterative algorithms performing an orthogonal projection of a non-Hermitian input matrix operator in order to find its

eigenvalues and eigenvectors. This algorithm appears to be an efficient tool for finding a few eigenvalues in a standard or generalized eigenproblem from a chosen part of the operator spectrum.

2.1 Presentation of the sequential algorithm

Given a matrix operator A, the Arnoldi factorization reduces this operator to a form of an upper Hessenberg matrix. This operation is performed in an iterative numerically stable process described by the following formula:

$$AV_k = V_k H_k + f_k e_k^T \tag{1}$$

where:

A is the input n x n matrix,

H_k is a k x k upper Hessenberg matrix ($k < n$),

V_k is an n x k matrix whose columns are Arnoldi vectors,

f_k is a residual vector of size n, satisfying the relation $V_k^T f_k = 0$.

The idea of Arnoldi factorization is illustrated in Figure 1.

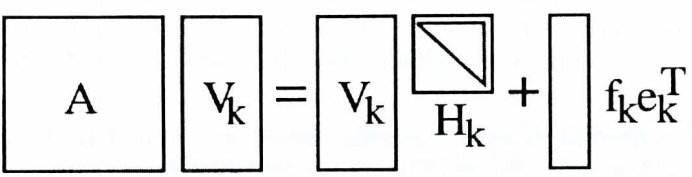

Figure 1. The schematic of Arnoldi factorization

The following similarity relation holds for the matrices H_k and A:

$$H_k = V_k^T A V_k \tag{2}$$

It has also to be noted here that, due to the characteristics of the method, the set of Arnoldi vectors v_i ($\{v_i\}_{i=1,\ldots,k} = V_k$) forms an orthonormal basis in the Krylov subspace K_k, where:

$$K_k = Span\{v, Av, A^2v, \ldots, A^{k-1}v\} \tag{3}$$

and $v \in R^n$ ($v \in C^n$). The orthonormal basis $\{v_i\}_{i=1,\ldots,k}$ is formed in k iterations of the basic Arnoldi algorithm. In general, the steps of the Arnoldi algorithm (or more precisely the Arnoldi modified Gram-Schmidt algorithm) are given as follows:

1. Choose an initial vector v_1 such that $\|v_1\|_2 = 1$

2. Iterate: For j=1,2, ..., k do:

(a) $w := Av_j$

(b) For i=1, 2, ..., k do:

- $h_{ij} = (w, v_i)$,
- $w := w - h_{ij}v_i$

(c) $h_{j+1,j} = ||w||_2$

(d) $v_{j+1} = w / h_{j+1,j}$

It has to be noted that during the factorization process the information on the input matrix (A) is passed to the algorithm only via the matrix-vector product Av_j. This feature of Arnoldi method is used in the discussed P_ARPACK library by introducing a very convenient reverse communication interface which enables the user to choose an adequate way to perform the Av_j operation. During the iteration the (j+1)-th vector is formed from the previous one (v_j) by: Av_j multiplication (step 2(a) of the algorithm), orthogonalization (2(b)) and normalization (2(c), (d)). As the vector f_k becomes small enough the eigenvalues of H_k start to approximate the eigenvalues of A and the quality of the approximation may be found by calculating the residual norm: $|| Av_i - \lambda_i v_i ||_2$. Once the norm falls below a desired level of accuracy, the algorithm terminates. One of the disadvantages of this basic Arnoldi algorithm which emerges at this point of the discussion is that the number of iterations (k) is not pre-determined for a fixed accuracy of the eigenvalues approximation. Consequently both time and memory complexity of the problem is not well defined and may increase significantly as the number of iterations becomes large. Another characteristic of the algorithm is that it initially converges to the eigenvalues with the largest magnitude.

Therefore modifications of the basic Arnoldi algorithm have been proposed by Saad ([6]) and Sorensen ([2] and [3]). These modifications are based on the idea of restarting the iterative Arnoldi process after e.g. k steps with an updated initial vector v_1. As the number k remains fixed also memory requirements and numerical complexity become pre-determined. Another modification enables the user of the library to find the eigenvalues from a specified part of the spectrum by filtering out the 'unwanted' eigenvalues before each restart, using e.g. polynomial filters, so that the iterative process converges only to 'wanted' eigenvalues. A detailed presentation of the problem of modifying the operator's spectrum by applying polynomial filters may be found in the article by Sorensen ([3]).

2.2 Parallelization of the algorithm in a distributed memory environment

A straightforward parallelization method of the Arnoldi algorithm has been proposed by Maschhoff and Sorensen ([1]) and implemented in the P_ARPACK library designed for distributed memory parallel systems. It is based on the data distribution programming paradigm applied to the operation of Arnoldi factorization. Due to the reverse communication interface the problem of parallelizing the calculation of the matrix-vector product Av_j (cf. previous paragraphs), which often becomes a non-trivial task, is left to the user. Consequently the implementors of

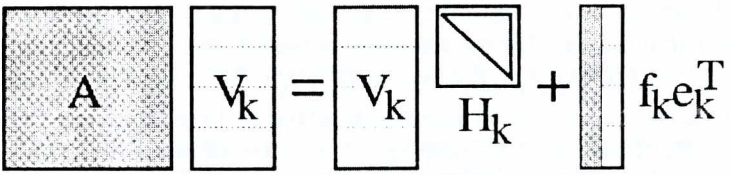

Figure 2. Block data distribution among the processors during the Arnoldi factorization in the P_ARPACK library

P_ARPACK need to handle only the parallel execution of Arnoldi iteration with Av_j vector as an input provided by multiple processors.

If once again the formula for the factorization is examined:

$$A V_k = V_k H_k + f_k e_k^T \tag{4}$$

with the same meaning of the symbols as in the previous paragraphs, then the parallelization scheme illustrated in Figure 2 is as follows:

- the k x k upper Hessenberg matrix H_k is replicated on every processor,
- the matrix V_k is block-distributed across a one-dimensional processor grid (as shown in Figure 2),
- f_k and workspace are distributed accordingly,
- the representation of the input matrix A in a distributed memory system is to be chosen by the user according to his or her needs. Typically the block-distribution of V is commensurate with the parallel decomposition of the matrix A.

The user of P_ARPACK is provided with a Single Program Multiple Data (SPMD) template which enables a simple and consistent handling of parallel operations inside the programs. Because of this the SPMD style becomes dominant or even standard in parallel programming. From the point of view of P_ARPACK interface the SPMD style is also very convenient as calling of library routines in a parallel program looks virtually the same as in a serial code. The differences between calling P_ARPACK and ARPACK routines include e.g. passing a local size of the problem n_{loc} instead of a global size n or a block of V_k matrix local for every processor instead of the entire matrix as it is in the serial case. The P_ARPACK routines need also to be supplied by the user with information on message-passing environment such as the MPI communicator (e.g. MPI_COMM_WORLD communicator) or the BLAS context.

A crucial aspect of parallelization in a distributed memory environment is the amount of communication occurring between the processors during the run-time of the algorithm. Luckily in the parallel version of the Arnoldi algorithm there are only two communication points within the Arnoldi factorization operation. One of them is the computation of the norm of the distributed vector f_k and the other is the orthogonalization of f_k to V_k using modified Gram-Schmidt process. If once again the

serial Arnoldi factorization algorithm is considered (cf. previous paragraph) then it is clearly seen that the step 2(c) (normalization) requires the computation of a 'global' norm that must be known to all the processors. Step 2(b) requires the computation of a global sum $\Sigma (w, v_i)v_i$ as the different parts of the do loop (associated with different ranges of values of indices i) are performed on different processors. This global sum has to be known by all the processors during the factorization in order to be able to find the next approximation of the matrix V and the next form of the replicated upper Hessenberg matrix H_k. In case of the MPI implementation of the library the communication operations are performed by using global reduction functions such as MPI_ALLREDUCE. This is entire communication which is done by the library routines, but some additional communication will always occur while calculating the user-dependent matrix-vector product Av_j (step 2(a)).

A certain kind of trade-off may be observed in the parallelization strategy applied to Arnoldi factorization. As the form of the upper Hessenberg matrix is calculated by each processor the communication between them is not needed. Obviously all the processors calculate the same thing introducing some redundancy to the algorithm which may lead to a serial bottleneck as the size of H_k increases. This may eventually lead to the lack of scalability of the method.

2.3 Calling parallel ARPACK routines

The user of P_ARPACK software is obliged to follow certain rules while constructing a parallel program. These include writing the code in the SPMD style and also inserting a reverse communication loop as a basic means of communication with the discussed numerical library. Figure 3 shows a fragment of a Fortran77 code of a parallel program using P_ARPACK library routine `pdnaupd()` in order to solve an eigenproblem of a given non-symmetric real matrix A. The code assumes that the inter-process communication is handled by the Message Passing Interface (MPI) library.

In the Figure 3 it is seen that the MPI_COMM_WORLD communicator is used as a communication space among processors. The number of processors which execute the program is determined in the next line. Following is an initialization of various parameters needed by the `pdnaupd()` routine, including e.g. both global and local problem size, the number of eigenvalues to be found or the type of the eigenproblem (standard/generalized). Also an initial vector is specified during the initialization. The *tol* parameter in Figure 3 determines the stopping criterion for the Arnoldi factorization (cf. equation (1)). The algorithm stops if the condition:

$$||Au_i - u_i\lambda_i||_2 \leq tol \cdot |\lambda_i| \tag{5}$$

is satisfied for all λ_i. Other parameters define various options of the algorithm including the maximum number of Arnoldi updates allowed or types of shifts used. A detailed description of all the parameters of P_ARPACK routines may be found in [7].

```
c --------- Parameter selection for pdnaupd() -------------
      comm = MPI_COMM_WORLD ! Set the communicator
      call MPI_Comm_size(comm,  ! Determine the number of
                nprocs, ierr)   ! processors used
      n     = N       ! size of the problem
      nev   = NEV     ! number of eigenvalues to be computed
      ncv   = NCV     ! number of orthogonal columns of V
      nloc = n/nprocs ! Determine local size of the problem
      bmat  = 'I'     ! standard eigenvalue problem
      which = 'LM'    ! find eigenvalues with largest magn.
      tol    = 1.e-8  ! set the desired accuracy
      ido    = 0      ! first call to reverse communication
      info   = 1      ! resid contains the initial vector
      do 100 i = 1, nloc ! initialize resid as a vector
         resid(i) = 1.d0 ! with 1's as all the elements
 100  continue
      iparam(1) = 1    ! exact shifts with respect to H
      iparam(3) = 1000 ! maximum number of updates
      iparam(7) = 1    ! Mode set to 1
c ------------------ Reverse communication loop ----------
 200  continue
         call pdnaupd(comm, ido, bmat, nloc, which, nev,
     &               tol, resid, ncv, v, ldv, iparam,
     &       ipntr, workd, workl, lworkl, info )
c
         if (ido .eq. -1 .or. ido .eq. 1) then
c           Compute  matrix-vector product: A*v
            call Av(nloc, workd(ipntr(1)), workd(ipntr(2))
c
            go to 200 ! Loop back to call pdnaupd() again
         endif
c ------------------------------------------------------
```

Figure 3. Calling `pdnaupd()` P_ARPACK subroutine in a reverse communication loop

Figure 3 also shows a sample reverse communication loop in which the `pdnaupd()` routine is called and if the stopping criterion has not been satisfied then a user-supplied matrix-vector multiplication subroutine is called. A similar construction of the main loop was used in all the programs used by the author to test the performance of P_ARPACK subroutines.

3. Parallel programming in the IBM SP2 system

As already mentioned all the calculations and tests of the parallel Arnoldi package which are described later in this article have been performed in the 16-processor IBM SP2 system (IBM 9076 Scalable POWER2 parallel system) at the Academic Computer Centre TASK in Gdańsk. Quite often knowing the specifics of the parallel system environment is crucial to obtaining a certain grade of performance. Therefore some important details concerning methods of compilation and running of the programs are presented below.

3.1 Tools supporting message-passing programming in the SP2 environment

The IBM SP2 system contains several tools for compiling and executing parallel programs based on the message-passing paradigm, including Parallel Environment (PE) with Parallel Operating Environment (POE) as well as LoadLeveler batch job scheduler designed for the POWER2 architecture.

The IBM's Parallel Environment (PE) is designed for the development and execution of parallel programs written in Fortran or C (C++) languages. The Parallel Environment (PE) consists of several parts, such as message-passing libraries e.g. Message Passing Interface (MPI) [5] (IBM's proprietary implementation of the popular portable library) or Message Passing Library (MPL) as well as the software package called Parallel Operating Environment (POE) which appears to be indispensable while compiling, running, and monitoring parallel programs. POE provides compiler scripts (such as mpxlf - message passing Fortran compiler) that automatically link in the message passing libraries when a parallel program is compiled, and environment variables that allow to control the run-time environment. At run time POE is responsible for allocating nodes to the job, reproducing the local environment on each remote node, loading the communication libraries and eventually executing the program. Detailed guide to using the POE environment may be found in the Cornell Theory Center web site ([9]).

3.2 Running parallel jobs in the IBM SP2 system

Basically there are two methods of running parallel programs in the IBM SP2 system. The first is running the parallel jobs interactively, providing the user with opportunity to monitor the progress of the program at run time. The second method is running batch jobs. In fact it is the only reasonable technique for executing large, computationally intensive jobs in an efficient way. A tool which supports running batch jobs in the IBM SP2 system is LoadLeveler - a job scheduler that distributes jobs to nodes, providing load-balancing of all the nodes of the system. The scheduler decides when and how a batch job is run based on preferences set up by the user and system administrator. In the next section some essential details on running parallel jobs with use of LoadLeveler are described.

4. Numerical results

After presenting the software tools for parallel programming in the IBM SP2 at the Academic Computer Centre TASK in Gdańsk some results of tests of P_ARPACK subroutines are given in this part of the article. These include benchmarks of P_ARPACK routines discussing their scalability and issues associated with serial bottlenecks and communication overheads.

4.1 Compiler options and LoadLeveler scripts

All the codes used during the performance tests written in Fortran77 and MPI library were compiled with the standard xlf IBM's XL Fortran compiler available in AIX based systems. In order to simplify compiling and linking of message-passing programs the mpxlf compiler script, provided within the Parallel Operating Environment (POE) has been used. This script automatically links in the necessary message-passing libraries and sets environment variables which allow the user to control the run-time environment.

During the tests the programs were compiled with different levels of compiler optimizations (`-O2` = `-O` - the standard optimization and `-O3` - high level of optimization). Another option that was used - `-qhot` indicated that the compiler should determine whether or not to perform high order transformations on loops during the optimization with `-O3` flag. Another option: `-qarch=pwr2` ensured generating a code tuned for POWER2 architecture. The programs were linked dynamically to ARPACK, PARPACK and ESSL (version for POWER2 system) libraries. The ESSL library (`-lesslp2` linker flag) provided a set of BLAS subroutines used by ARPACK that are tuned for the SP2 architecture. The Message Passing Interface library linked to the codes was also the IBM's proprietary implementation provided within POE. According to the available documentation the presented set of libraries chosen for linking to the testing programs was about to ensure the optimum performance in the IBM SP2 distributed memory system. Still, not all the parameters affecting the run-time performance of the programs may be specified by the compiler and linker flags. Some of them are defined in the scripts necessary to run the jobs in parallel in the IBM SP2 environment.

Although parallel programs in the SP2 system may run either as interactive or batch jobs, the latter method was used as it offers a more stable and predictable environment for doing performance tests. While running jobs interactively it was found out that the different measurements of execution times of identical programs and input data varied by hundreds percent. Therefore it has been decided that all the programs were to be executed as batch jobs using the IBM's LoadLeveler job scheduling system. Later on it resulted that some additional effort has to be made in order to ensure more reliable timings.

The LoadLeveler command file specifies all the parameters necessary to run a batch job. Figure 4 presents such a script used during the tests. The most important parameter in this script is the `-euilib xx` option specifying the communication subsystem (CSS) used for message-passing during the execution of the program. As already mentioned in the previous sections, the SP2 system offers two CSS libraries, namely the IP (Internet Protocol) and US (User Space). The `-euilib us` specifies the US and `-euilib ip` chooses the IP. Additionally while choosing the IP based inter-processor communication, the physical medium of communication has also to be specified by putting the `-euidevice css0` for High Performance Switch (HPS) or `en0` for Ethernet adapter. Obviously High Performance Switch offers by

far a better message-passing performance than the Ethernet link between the processors. Moreover while using Ethernet it becomes unrealistic to obtain any reliable results, as the system uses this network for many different purposes, including e.g. the communication with outer environment which considerably degrades the performance. Consequently only the High Performance Switch has been used as the communication medium. The command file also specifies the number of processors and adequate processor pool to be used by the job and determines where the output should be redirected.

```
#############################################################
#
# @ environment = "LL_JOB=TRUE"
# @ executable = /usr/bin/poe
# @ arguments =/users/mrewiens/Tests/testscript -euilib us
# @ min_processors = 8
# @ max_processors = 8
# @ initialdir = /usrp1/mrewiens/Tests
# @ output = test.$(Cluster).us.out
# @ error = test.$(Cluster).us.err
# @ job_type = parallel
# @ requirements = (Adapter == "hps_user" && Pool == 2)
# @ class = POE
# @ queue
#
#############################################################
```

Figure 4: The LoadLeveler command file test.us.cmd.

```
########################################

cd /tmp
cp /users/holk1/mrewiens/Tests/test_SP2 .
/tmp/test_SP2

########################################
```

Figure 5. The script passed as an argument for the poe command

In the `test.us.cmd` file a name of the script (`testscript`) is specified. This script file (cf. Figure 5) enables the poe to run the actual executable on every specified node of the system. One very important thing about this script should be noted. Before executing the program (`/tmp/test_SP2`) the executable is copied by each of the processors to their local disks from the author's home directory. This operation prevents from use of NFS-mounted disks, which offer a rather unpredictable response during the run-time and therefore becomes essential while measuring execution times of the programs. It has been observed that if the executables

were not copied to local disks the measured time varied by up to a 100 per cent which made the results clearly unacceptable. Such behaviour of the SP2 system has also been reported by Allan et. al. ([8]). This implies that the executables should always be copied to local disks of the processors in order to obtain a fairly stable execution time.

4.2 Run-time performance of P_ARPACK subroutines

The aim of the first series of tests of P_ARPACK subroutines performed in the IBM SP2 system was to find out what is the influence of the compiler optimization options and versions of subroutines of BLAS library used by ARPACK on the performance of the examined library.

The PARPACK subroutine tested was `pdnaupd()` which performs the Arnoldi iteration for the non-symmetric real problems. The operator with unknown eigenvalues was the square diagonal matrix with random elements between 0 and 1 located on the diagonal. Four of the elements of the spectrum were incremented by 1.1. This allowed them to be found easily by the `pdnaupd()` routine. The matrix had the size of 160.000 and was block-distributed among the processors. The `pdnaupd()` routine was called with the following parameters: NEV=4 (number of requested eigenvalues =4), NCV = 20 (number of columns of the vector V =20) and WHICH='LM' (eigenvalues with the largest magnitude were to be found). The characteristics of the problem presented above have been chosen so that it is independent of the changing of the problem size and the number of Arnoldi update iterations remains constant in every case. It may be said that the possible problem-specific factors that could influence the performance of P_ARPACK library subroutines have been eliminated.

The described series of tests consisted of compiling the P_ARPACK library with different options and running the test program described above, compiled with the same flags as the library. The following different sets compiler directives have been used to construct both P_ARPACK library and the executable:

- `mpxlf -O2 -qarch=pwr2 xxx.f -o xxx` (Table 1)
- `mpxlf -O3 -qarch=pwr2 xxx.f -o xxx` (Table 2)
- `mpxlf -O3 -qarch=pwr2 xxx.f -o xxx -lesslp2` (Table 3)
- `mpxlf -O3 -qhot -qarch=pwr2 xxx.f -o xxx -lesslp2` (Table 4)

Tables 1, 2, 3 and 4 show the results of performance tests for different compilation flags used while building both the library and the tested executable. In the tables all the times are average user times (not system or wall-clock times) and are given in seconds. The time measured in all the cases is the time spent in the Arnoldi iteration of the `pdnuapd()` routine. They should be regarded as total times needed by the IRAM algorithm to converge to the wanted solutions. During the tests different communication subsystems (CSS) have been used: the column 'Time (ip)' shows the

timings while the Internet Protocol has been used for communication, while 'Time (us)' gives times measured while the User Space protocol has been applied. In all the cases during this series of tests the number of Arnoldi updates (when the Arnoldi algorithm restarts with an updated initial vector v_1) equaled 4 and did not change with the changing number of processors used. In Figure 6 the results of the tests are given in the form of a graph showing the execution times of the test application for different compilation methods as a function of the number of processors used during the execution.

Number of nodes	Time (ip)	Speedup (ip)	Time (us)	Speedup (us)
1	51.05	1.00	51.17	1.00
2	25.59	1.99	26.78	1.91
4	17.64	2.89	20.10	2.55
8	7.71	6.62	9.83	5.20

Tabl e 1. Performance of `pdnaupd ()` for ip and us libraries, N = 160000, NEV = 4, NCV = 20, number of Arnoldi iterations = 4. Compiler directive: `mpxlf -O2 -qarch=pwr2, BLAS = standard`. All times are given in seconds

Number of nodes	Time (ip)	Speedup (ip)	Time (us)	Speedup (us)
1	72.96	1.00	77.25	1.00
2	38.52	1.89	39.96	1.93
4	25.33	2.88	25.43	3.04
8	12.73	5.73	12.91	5.98

Table 2. Performance of `pdnaupd ()` for ip and us libraries, N = 160000, NEV = 4, NCV = 20, number of Arnoldi iterations = 4. Compiler directive: `mpxlf -O3 -qarch=pwr2, BLAS = standard`. All times are given in seconds

Number of nodes	Time (ip)	Speedup (ip)	Time (us)	Speedup (us)
1	72.27	1.00	77.03	1.00
2	38.26	1.89	39.72	1.94
4	21.82	3.31	29.23	2.63
8	13.35	5.41	13.09	5.88

Table 3. Performance of `pdnaupd ()` for ip and us libraries, N = 160000, NEV = 4, NCV = 20, number of Arnoldi iterations = 4. Compiler directive: `mpxlf -O3 -qarch=pwr2, BLAS = ESSL`. All times are given in seconds

Number of nodes	Time (ip)	Speedup (ip)	Time (us)	Speedup (us)
1	76.77	1.00	77.66	1.00
2	38.18	2.01	39.26	1.95
4	19.62	3.91	21.60	3.55
8	10.78	7.12	10.05	7.63

Table 4. Performance of `pdnaupd ()` for ip and us libraries, N = 160000, NEV = 4, NCV = 20, number of Arnoldi iterations = 4. Compiler directive: `mpxlf -O3 -qhot -qarch=pwr2, BLAS = ESSL`. All times are given in seconds

The results of the tests appear to be rather surprising, as for the considered size of the problem N = 160000, the total time spent in the Arnoldi iteration is shorter if the library is compiled with a lower level of optimization, i.e. the -O2 flag and not

-O3 flag. While using only the -O3 flag the performance degrades by about 30%. This kind of situation has also been observed by Allan et. al. ([8]) and is due to some code-specific characteristics. Still if the US communication subsystem is used then the speed-up in the execution time is higher if the -O3 flag is used. It also has to be noted that this is not the case while applying the IP CSS.

The comparison between using the IP and US communication subsystem does not give univocal conclusions either. As it can be found from the tables the measured times are usually larger in the case of the US protocol used. This is surprising as the User Space protocol has been designed especially for the SP2 system providing a more efficient inter-processor communication. Still, the use of the US protocol may result in a better performance while running large programs with an intensive inter-processor communication. (As already noted the parallel Arnoldi method does not require much communication to be performed during the factorization.) One positive aspect of the use of US CSS which can be inferred from the output data is that while the -O3 flag is used the speed-up is better than for the IP CSS. This result may confirm that the US CSS provides a more efficient communication between the nodes of the SP2 system.

Table 4 shows results of performance tests if an additional compiler flag `-qhot` is used during the compilation of the P_ARPACK library. This flag forces the compiler to determine whether or not to perform high level optimization (`-O3`) on specific loops in the program's code. Consequently different parts of the code as optimized with different optimization levels. Although the measured times are similar to those obtained only with `-O3` flag, the speed-up obtained in this compilation method appears to be the highest, e.g. for 8 processors the speed-up exceeds 7 (cf. Table 4), while in the earlier cases it does not reach 6 (cf. Tables 1, 2, 3).

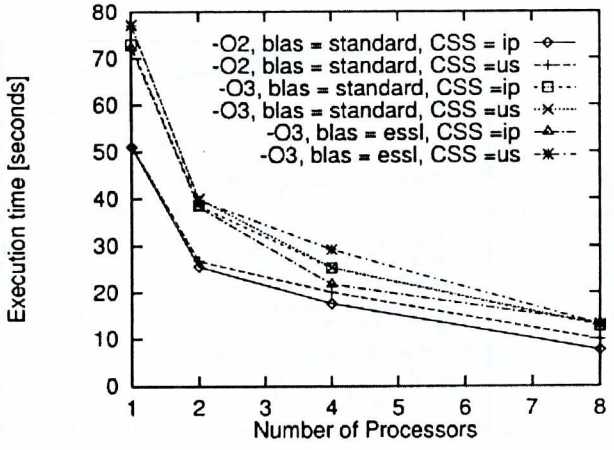

Figure 6. Execution time (in seconds) of `pdnaupd()` routine for the problem size N = 160000, number of Arnoldi iterations = 4, NEV = 4, NCV = 20. The results in the graph show timings for library routine compiled with different levels of optimization (`-O2, -O3`) and linked to different versions of BLAS library (BLAS version 2 (standard) and IBM's implementation of BLAS included in ESSL library)

Another issue that has been examined in the above tests was the use of BLAS library functions by the P_ARPACK library. Parallel ARPACK software uses a number of Basic Linear Algebra Subroutines including an matrix-vector product subroutine `Xgemv` or a matrix by upper triangular matrix multiplication `Xtrmm`. In the first test (Table 2) a BLAS version 2 routines provided together with ARPACK have been used, while in the second test (Table 3) the implementations of BLAS subroutines from the IBM's ESSL library were linked to ARPACK routines. The tests show that the performance of the `pdnaupd()` does not depend much on the version of BLAS library used, as minor differences in the execution times are observed. The authors of ARPACK still suggest ([7]) that the BLAS libraries already installed in the system should be used whenever possible instead of those provided with the ARPACK software. Consequently in all the following tests only the IBM's proprietary ESSL library containing BLAS subroutines has been used.

The next series of tests performed measured the scalability of the `pdanupd()` P_ARPACK library subroutine. The same program (`test.f`) as in the previous section was used during the measurements. As mentioned the input matrix A in the `test.f` has been chosen so that its characteristic is independent of the size of the problem. More precisely during the scalability tests with the `pdnaupd()` subroutine, the number of Arnoldi update iterations remained the same for a fixed number of eigenvalues to be computed (NEV = 4 = const.) and the changing size of the problem. The size of the problem was chosen to be $N = 200000$ for 1 processor and increased linearly with the number of processors. Both IP and US communication subsystems have been used during the measurements and the compiler directive used during the compilation of both the P_ARPACK software and the testing program was: `mpxlf -O3 -qhot -qarch=pwr2 xxx.f -o xxx`. The results of the tests are given in the Table 5.

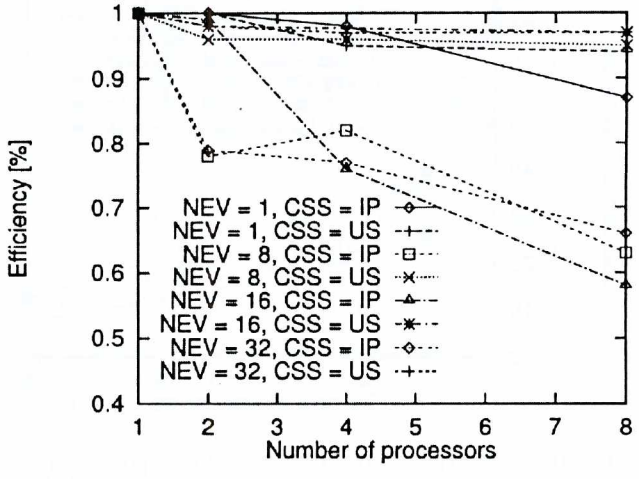

Figure 7. Efficiency of `pdnaupd()` P_ARPACK subroutine. The size of the test problem equals $N = 200000 * Number\ of\ nodes$. NEV = 1, 8, 16, 32, NCV = 40. The number of Arnoldi update iterations remained constant for a fixed number of eigenvalues (NEV) to be computed.

Number of nodes	Time (ip)	Efficiency	Time (us)	Efficiency
1	97.29	1.00	95.81	1.00
2	97.47	1.00	93.64	1.02
3	125.80	0.77	96.53	0.99
4	143.92	0.67	99.99	0.96
5	134.11	0.72	95.92	1.00
6	110.43	0.88	96.95	0.99
7	138.25	0.70	97.92	0.98
8	136.95	0.71	100.98	0.95

Table 5. Scalability of `pdnaupd()` routine. The size of the problem equals *N = 200000 * Number of nodes*, NEV = 4, NCV = 20, CSS = ip or us, compiler directive: `mpxlf -O3 -qhot -qarch=pwr2 test.f`.

The results of scalability tests show that the efficiency of the `pdnaupd()` P_ARPACK subroutine remains relatively high for the considered size of communicators (number of system nodes). It is well seen that application of the US communication subsystem gives a considerable improvement in scalability and efficiency of the executed program. For the number of processors from 1 to 8 and the problem size N = 200000, ..., 1600000 the efficiency stays above the level of 95 %, while in case of the IP CSS it falls below 70 %. The reason for such behaviour is that for large problem sizes, the amount of communication increases, so that the transmission rate of messages between the processors (which is much higher for the US CSS - cf. [8]) starts to play an important role during the run-time. It has to be noted that the above results stay in a close accordance with the scalability results given in ([1]) for tests performed in the Maui HPCC SP2 machine. Obviously scalability is not perfect. This effect is due a serial bottleneck caused by the algorithm redundancy while replicating the upper Hessenberg matrix by all the processors during the Arnoldi factorization.

The tests have also been performed using the previously described code in order to find out the dependence of the total execution time of the `pdnaupd()` P_ARPACK subroutine on the number of eigenvalues (NEV) to be computed. In the IRAM algorithm the increment of NEV=k causes an increment in the memory storage requirements determined by the upper Hessenberg matrix size ($k \times k$) and the size of the vector ($k \times n$). In the parallel implementation of ARPACK these increments may greatly affect performance and scalability of the library routines. As already mentioned the upper Hessenberg matrix is replicated on every processor and therefore may cause a serial bottleneck as its size increases. An increasing value of k may also result in the increased communication costs, e.g. during the re-orthogonalization phase where more global sums have to be computed and communicated using the global reduction operations.

Table 6 shows the timings obtained for the `pdnaupd()` subroutine, The size of the problem was equal *N = 200000 * Number of nodes*. The number of eigenvalues to be computed NEV = 1, 8, 16, 32 (NCV = 40). The number of Arnoldi update iterations remained constant for a fixed number of eigenvalues (NEV) to be computed. Once again two communication subsystems were considered.

It may be noticed that in case of using the IP communication subsystem the efficiency decreases faster for larger values of NEV, e.g. for 8 processors the efficiency equals: 0.87 (NEV = 1), 0.63 (NEV = 8), 0.58 (NEV = 16). (The graph of performance efficiency vs. the number of processors for different values of NEV is shown in Figure 7.) This effect has not been observed while the US protocol is used. A conclusion may be drawn that the degradation in the performance in the first case is mainly due to communication overhead and not a serial bottleneck caused by replicating the H_k matrix. If this had been the cause, the efficiency would have degraded for both communication subsystems.

Another issue which has been investigated during the tests described above were the times spent in the orthogonalization phase during the Arnoldi factorization for different numbers of eigenvalues to be computed and different numbers of processors used during the run-time (the parameters remained the same as those given in Table 6). During the orthogonalization phase global sums have to be computed and the number of these global reduction operations depends on the number of eigenvalues NEV to be computed.

It has been observed that the time spent on orthogonalization increases with the increasing number of processors used. This is due to the communication costs which appear during the computation of global sums. It has also been noticed that the increment of NEV caused a relative increment in the orthogonalization time. For instance: for the IP CSS the time spent on orthogonalization using 8 processors was 21% larger than the time of an analogous serial operation for NEV =1 and almost 300% larger for NEV = 8. The same effect occured for the US CSS, though the time increments were not so large.

NEV	Number of nodes	Time (ip)	Efficiency	Time (us)	Efficiency
1	1	40.96	1.00	40.92	1.00
	2	41.05	1.00	40.98	1.00
	4	41.90	0.98	42.68	0.95
	8	47.00	0.87	43.27	0.94
8	1	83.11	1.00	82.90	1.00
	2	105.87	0.78	85.93	0.96
	4	100.78	0.82	86.68	0.96
	8	131.61	0.63	87.26	0.95
16	1	140.69	1.00	143.14	1.00
	2	141.33	0.99	146.62	0.98
	4	185.00	0.76	*	*
	8	240.11	0.58	147.02	0.97
32	1	390.62	1.00	389.61	1.00
	2	496.79	0.79	395.85	0.98
	4	503.99	0.77	399.45	0.97
	8	586.86	0.66	401.66	0.97

Table 6. Performance of `pdnaupd()` for ip and us libraries, $N = 200000 * Number\ of\ processors$, NEV = 1, 8, 16, 32, NCV = 40, compiler directive: `mpxlf -O3 -qhot -qarch=pwr2 test.f`. All times are given in seconds

It is clearly seen in all the results that the US communication subsystem provided a faster and more efficient background for message-passing (for larger numbers of nodes and larger values of NEV).

It has to be explained here why in the measurements of the total time spent in the Arnoldi algorithm (for the US CSS) the efficiency did not decrease significantly (cf. Table 6). This can be done in the following way: Although the time spent on orthogonalization increases with the incerasing number of processors, the percentage of time actually spent in the orthogonalization phase decreases with an increasing value of NEV. Due to a relatively slow increment in the orthogonalization time for US CSS, the overall efficiency of `pdnaupd()` performance remains on the same level.

5 Conclusion

In this article issues concerning the implementation and performance of a parallel Implicitly Restarted Arnoldi Method (IRAM) in a distributed memory IBM SP2 system have been discussed. Various tests have been performed in order to find out the run-time characteristics of the P_ARPACK library subroutines. The following summary of conclusions can be given:

- P_ARPACK provides an efficient parallel implementation of the IRAM algorithm for a distributed memory system such as IBM SP2, offering a high scalability and speed-up during a parallel execution.
- The parallelization strategy in the P_ARPACK software appears to be well chosen as no serious serial bottlenecks have been observed during the performance tests. Also the communication overheads do not block the scalability of the method for a wide range of input parameters.
- The reverse communication technique as well as the Single Program Multiple Data (SPMD) template provide a user with a convenient and consistent interface to the P_ARPACK library simplifying message-passing programming.

Also some care should be taken while specifying the environment used in building and running message-passing programs. The following general guidelines can be given:

- A special care should be taken while choosing compiler flags, including different levels of optimizations.
- During the run-time the User Space (US) communication subsystem should be used as it provides a more stable performance of the applications, a better scalability and the fastest message-passing background.
- The Message Passing Interface (MPI) library to be used should be the IBM's proprietary library available within the Parallel Operating Environment (POE). Still, the usefulness of IBM's numerical libraries (ESSL) implementing BLAS subroutines together with P_ARPACK remains an open question.

Reference

[1] K. J. Maschhoff, D. C. Sorensen, *P_ARPACK: An Efficient Portable Large Scale Eigenvalue Package for Distributed Memory Parallel Architectures*}, Rice University, available at: ftp.caam.rice.edu.

[2] D. C. Sorensen, *Implicitly Restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations*, Proceedings of an ICASE/LaRC Workshop, May 23-25 1994, Hampton, VA, D. E. Keyes, A. Sameh and V. Venkatakrishnan, eds., Kluwer, 1995

[3] 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

[4] M. P. Dębicki, P. Jędrzejewski, J. Mielewski, P. Przybyszewski, M. Mrozowski, *Application of the Arnoldi Method to the Solution of Electromagnetic Eigenporblems on the Multiprocessor Power Challenge Architecture*, Technical Report no. 19/95, Faculty of Electronics, Technical University of Gdańsk

[5] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, International Journal of Supercomputer Applications and High Performance Computing, 8(3/4), 1994

[6] Y. Saad, *Numerical Methods for large eigenvalue problems*, Manchester University Press Series in Algorithms and Architectures for Advanced Scientific Computing, 1992

[7] R. B. Lehoucq, D. C. Sorensen, C. Yang, *ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*, document available by anonymous ftp from: ftp.caam.rice.edu.

[8] R. J. Allan, M. F. Guest, *Parallel Application Software on High Performance Computers, 1 The IBM SP2 and Cray T3D, Technical Report CCLRC HPCI Centre at Daresbury Laboratory, Daresbury 1996*

[9] Cornell Theory Center web site: http://www.tc.cornell.edu