John von Neumann Institute for Computing

Basic numerical libraries for parallel systems

Inge Gutheil

published in

Modern Methods and Algorithms of Quantum Chemistry,
J. Grotendorst (Ed.), John von Neumann Institute for Computing, Jülich, NIC Series, Vol. 1, ISBN 3-00-005618-1, pp. 29-47, 2000.

http://www.fz-juelich.de/nic-series/

BASIC NUMERICAL LIBRARIES FOR PARALLEL SYSTEMS

I. GUTHEIL
John von Neumann Institute for Computing
Central Institute for Applied Mathematics
Research Centre Jülich, 52425 Jülich, Germany
E-mail: i.gutheil@fz-juelich.de

Three public domain libraries with basic numerical operations for distributed memory parallel systems are presented: ScaLAPACK, PLAPACK, and Global Arrays. They are compared not only with respect to performance on CRAY T3E but also to user-friendliness.

1 Introduction

There are many projects for parallelization of numerical software. An overview of public domain libraries for high performance computing can be found on the HPC-Netlib homepage[1]. Often these libraries are very specialized either concerning the problem which is treated or the platform on which they run. Three of the more general packages based on message-passing with MPI will be presented here in some detail: ScaLAPACK[2], PLAPACK[3], and Global Arrays[4].

Often there is an MPI-implementation on shared-memory multiprocessor systems, hence libraries based on MPI can also be used there and often very efficiently as message-passing programs take great care of data locality.

1.1 ScaLAPACK, Scalable Linear Algebra PACKage

The largest and most flexible public domain library with basic numerical operations for distributed memory parallel systems up to now is ScaLAPACK. Within the ScaLAPACK project many LAPACK[5] routines were ported to distributed memory computers using message passing.

The communication in ScaLAPACK is based on the BLACS (Basic Linear Algebra Communication Subroutines)[6]. There are public domain versions of the BLACS based on MPI and PVM available. For CRAY T3E there is also a version of the BLACS in Cray scientific libraries (libsci)[7] using Cray shared memory routines (shmem) which is often faster than the public domain versions.

The basic routines of ScaLAPACK are the PBLAS (Parallel Basic Linear Algebra Subroutines). They contain parallel versions of the BLAS[a], which are parallelized using BLACS for communication and sequential BLAS for computation. As most vendors offer optimized sequential BLAS the BLAS and PBLAS deliver very good performance on most parallel computers.

Based on BLACS and PBLAS ScaLAPACK contains parallel solvers for dense linear systems and linear systems with banded system matrix as well as parallel routines for the solution of linear least squares problems and for singular value

[a]BLAS 1 contains vector-vector operations, e.g. dotproduct, BLAS 2 matrix-vector operations, e.g. matrix-vector multiplication, BLAS 3 matrix-matrix operations, e.g. matrix-matrix multiplication

decomposition. Routines for the computation of all or some of the eigenvalues and eigenvectors of dense real symmetric matrices and dense complex hermitian matrices and for the generalized symmetric definite eigenproblem are also included in ScaLAPACK.

ScaLAPACK also contains additional libraries to treat distributed matrices and vectors. One of them is the TOOLS library, which offers useful routines for example to find out which part of the global matrix a local process has in its memory or to find out the global index of a matrix element corresponding to its local index and vice versa. Unfortunately these routines are documented only in the source code of the routines and not in the Users' Guide. Another library is the REDIST library which is documented in the ScaLAPACK Users' Guide. It contains routines to copy any block-cyclicly distributed (sub)matrix to any other block-cyclicly distributed (sub)matrix.

ScaLAPACK is a Fortran 77 library which uses C subroutines internally to allocate additional workspace. Especially the PBLAS allocate additional workspace.

1.2 PLAPACK, Parallel Linear Algebra PACKage

PLAPACK does not offer as many black-box solvers as ScaLAPACK but is designed as a parallel infrastructure to develop routines for solving linear algebra problems. With PLAPACK routines the user can create global matrices, vectors, and multiscalars, and he may fill them with values with the help of an API (Application Programming Interface). To make the development of programs easier and to get good performance PLAPACK includes parallel versions of most real BLAS routines and solvers for real dense linear systems using LU-decomposition and for real symmetric positive definite systems applying Cholesky-decomposition which operate on the global data.

PLAPACK is a C library with a Fortran interface in Release 1.2. Unfortunately up to now Release 1.2 does not run on CRAY T3E.

1.3 Global Arrays

Like PLAPACK Global Arrays supplies the user with global linear algebra objects and an interface to fill them with data. For the solution of linear equations there is an interface to special ScaLAPACK routines which can be modified to use other ScaLAPACK routines, too. For the solution of the full symmetric eigenvalue problem Global Arrays contains an interface to PeIGS[8].

Global Arrays is a Fortran 77 library which uses a memory allocator library for dynamic memory management.

2 Data Distributions

There are many ways to distribute data, especially matrices, to processors. In the ScaLAPACK Users' Guide many of them are presented and discussed.

All of the three libraries described here distribute matrices to a two-dimensional processor grid, but they do it in different ways and the user can more or less influence the way data are distributed. Users who only want to use routines from the libraries

don't have to care for the way data are distributed in PLAPACK and Global Arrays. The distribution is done automatically. To use ScaLAPACK, however, the user has to create and fill the local parts of the global matrix on his own.

2.1 ScaLAPACK: Two-dimensional Block-Cyclic Distribution

For performance and load balancing reasons ScaLAPACK has chosen a two-dimensional block-cyclic distribution for full matrices (see ScaLAPACK Users' Guide). First the matrix is distributed to blocks of size $MB \times NB$. These blocks are then uniformly distributed across the $NP \times NQ$ processor grid in a cyclic manner. As a result, every process owns a collection of blocks, which are contiguously stored in a two-dimensional "column major" array.

This local storage convention allows ScaLAPACK software to efficiently use local memory by calling BLAS 3 routines on submatrices that may be larger than a single $MB \times NB$ block. Figure 1 shows the distribution of a 9×9-matrix subdivided into blocks of size 3×2 distributed across a 2×2-processor grid.

	0		1		0		1		0
0	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
1	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
0	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
	a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
	a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

Figure 1. Block-cyclic 2D distribution of a 9×9-matrix subdivided into 3×2-blocks to a 2×2-processor grid. The numbers outside the matrix indicate processor row and column indices respectively.

2.2 PLAPACK: "Physically based Matrix Distribution"

For those who want to develop programs which operate on PLAPACK distributed vectors and matrices it may be interesting to know more about the way data are distributed.

The distribution of matrices is induced by the distribution of vectors in a linear system $A\vec{x} = \vec{y}$: Vectors are divided into blocks of length NB, where NB is a blocking factor chosen by the user. These blocks are distributed to the processor grid in "column-first-order", i.e. processor (0,0) gets the first block, processor (1,0) the second one and so on.

The distribution of the matrix is now induced by requiring a column of matrix A to be assigned to the same column of processors as the corresponding element of $\vec{x}$ and the rows of A to the same row of processors as the corresponding element of $\vec{y}$,

e.g. processor (1,0) has x_2 and y_2, consequently the second row of A is distributed to the second row of processors and the second column of A is distributed to the first processor column.

Figure 2 first shows a 2×3-processor-grid and then a vector of length 7 distributed to it in "column-first-order" with block size 1. Below there is shown how a 7×7-matrix is distributed to the processor grid accordingly.

2×3
processor-grid

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)

distribution of the vector,
$NB = 1$

$$
\begin{aligned}
x_1 &\rightarrow (0,0)\\
x_2 &\rightarrow (1,0)\\
x_3 &\rightarrow (0,1)\\
x_4 &\rightarrow (1,1)\\
x_5 &\rightarrow (0,2)\\
x_6 &\rightarrow (1,2)\\
x_7 &\rightarrow (0,0)
\end{aligned}
$$

induced distribution of the matrix

		0			1		2	
		1	2	7	3	4	5	6
0	1	a_{11}	a_{12}	a_{17}	a_{13}	a_{14}	a_{15}	a_{16}
	3	a_{31}	a_{32}	a_{37}	a_{33}	a_{34}	a_{35}	a_{36}
	5	a_{51}	a_{52}	a_{57}	a_{53}	a_{54}	a_{55}	a_{56}
	7	a_{71}	a_{72}	a_{77}	a_{73}	a_{74}	a_{75}	a_{76}
1	2	a_{21}	a_{22}	a_{27}	a_{23}	a_{24}	a_{25}	a_{26}
	4	a_{41}	a_{42}	a_{47}	a_{43}	a_{44}	a_{45}	a_{46}
	6	a_{61}	a_{62}	a_{67}	a_{63}	a_{64}	a_{65}	a_{66}

Figure 2. Physically based distribution of a 7×7-matrix with block size 1 to a 2×3-processor-grid. The outmost numbers indicate processor row and column indices respectively, the next ones are matrix row and column indices.

Thus processor (0,0) has elements of rows 1, 3, 5, and 7 and columns 1, 2, and 7 because elements 1, 3, 5, and 7 of the vector are assigned to processor row 0 and elements 1, 2, and 7 of the vector are assigned to processor column 0.

2.3 Global Arrays: Two-dimensional Block Distribution

The global objects in this library are distributed in a very simple way and the user has little influence on that. Matrices are distributed into contiguous blocks and each process gets one of these blocks. The user can only choose the minimum number of rows or columns in a block. With this he can force for example a column block distribution by setting the minimum number of rows per block to the total number of rows of the global matrix.

When routines from other libraries are called there is an interface where data are redistributed in the way the other library expects them.

3 User-Interfaces

The user-interface of a library influences the decision for or against it. An easy-to-use interface can significantly reduce parallelization time.

3.1 ScaLAPACK

ScaLAPACK as a parallel successor of LAPACK attempts to leave the calling sequence of the subroutines unchanged as much as possible in comparison to the corresponding sequential subroutine from LAPACK. The user should have to change only a few parameters in the calling sequence to use ScaLAPACK routines instead of LAPACK routines.

Therefore ScaLAPACK uses so-called descriptors, which are integer arrays containing all necessary information about the distribution of a matrix. This descriptor appears in the calling sequence of the parallel routine instead of the leading dimension of the matrix in the sequential one.
For example the sequential BLAS 3 routine for the computation of $C = \alpha AB + \beta C$, A an $M \times K$-matrix, B a $K \times N$-matrix, overwriting the original C with the result, has the following calling sequence:

```
...
CALL SGEMM(TRANSA,TRANSB,M,N,K,alpha,A(1,1),LDA, &
           B(1,1),LDB,beta,C(1,1),LDC)
...
```

whereas the ScaLAPACK routine PSGEMM is called

```
...
! Call of PSGEMM with descriptors and the global
! starting indices of the whole matrix
CALL PSGEMM(TRANSA,TRANSB,M,N,K,alpha,A,1,1,DESCA, &
           B,1,1,DESCB,beta,C,1,1,DESCC)
...
```

Instead of taking the whole matrix starting with $A(1,1)$, any contiguous submatrix starting with $A(I,J)$, I and J global indices, can be multiplied with a submatrix of B starting with $B(J,L)$ by calling

```
...
CALL PSGEMM(TRANSA,TRANSB,M-I+1,N-L+1,K-J+1,alpha,A,I,J,DESCA, &
           B,J,L,DESCB,beta,C,I,L,DESCC)
...
```

The main problem is that the user has to take care of the data distribution. He has to choose the processor grid by initializing MP, the number of processor rows, and NP, the number of processor columns and to determine the blocking by choosing MB and NB, the number of rows and the number of columns per block, respectively. For many routines, especially for the eigenvalue solvers and the Cholesky decomposition, $MB = NB$ is necessary.

The conversion of global to local indices and vice versa is supported only by some auxiliary routines in the TOOLS sublibrary. It is completely left to the user to put the correct local part of the matrix to the right places and to put the correct data to the descriptor. The Users' Guide and the comments at the beginning of all routines are sufficient to use ScaLAPACK correctly but for someone not familiar with parallel programming it can be rather difficult and time-consuming to learn how to use it.

The main steps the user has to perform for creating and filling a matrix A are (it is assumed that MB=NB and N=M=K):

```
...
! Create the MP * NP processor grid
CALL BLACS_GRIDINIT(ICTXT,'Row-major',MP,NP)
! Find my processor coordinates MYROW and MYCOL
! NPROW should return same value as MP,
! NPCOL should return same value as NP
CALL BLACS_GRIDINFO(ICTXT, NPROW, NPCOL, MYROW, MYCOL)
! Compute local dimensions with routine NUMROC from TOOLS
! N is dimension of the matrix
! NB is block size
MYNUMROWS = NUMROC(N,NB,MYROW,0,NPROW)
MYNUMCOLS = NUMROC(N,NB,MYCOL,0,NPCOL)
! Local leading dimension of A,
! number of local rows of A
MXLLDA = MYNUMROWS
! Allocate only the local part of A
ALLOCATE(A(MXLLDA,MYNUMCOLS))
! Fill the descriptors, P0 and Q0 are processor coordinates
! of the processor holding global element A(0,0)
CALL DESCINIT(DESCA,N,N,NB,NB,P0,Q0,ICTXT,MXLLDA,INFO)
! Fill the local part of the matrix with data
do j = 1, MYNUMCOLS, NB   ! Fill the local column blocks
   do jj=1,min(NB,MYNUMCOLS-j+1) ! All colums of one block
      jloc = j-1 + jj   ! local column index
      jglob = (j-1)*NPCOL + MYCOL*NB +jj ! global column index
      do i = 1, MYNUMROWS, NB ! The local row blocks in this column
         do ii=1,min(NB,MYNUMROWS-i+1) ! The rows in this row block
            iloc = i-1 + ii   ! local row index
            iglob = (i-1)*NPROW + MYROW*NB + ii ! global row index
            A(iloc,jloc) = function of global indices iglob, jglob
         enddo
      enddo
   enddo
enddo
...
```

The four nested loops show how local and global indices can be computed from block sizes, the number of rows and columns in the processor grid and the processor

coordinates.

3.2 PLAPACK

The calling sequences of PLAPACK routines are also very similar to the ones of BLAS and LAPACK, e.g. the matrix-matrix-multiplication routine PLA_Gemm is called in the following way:

```
...
/* Call PLA_Gemm with global objects */
PLA_Gemm ( PLA_NO_TRANSPOSE, PLA_NO_TRANSPOSE, alpha,
           A, B, beta, C );
...
```

With PLAPACK always the whole global matrix is treated as the sizes and the distribution are implicitly contained in the global object. If the user wishes to deal with a submatrix only he has to create a so-called view into the matrix which is a new distributed object using the data and the memory locations of the whole matrix.

To write to or read entries from the global linear algebra objects of PLAPACK there is an Application Program Interface (API) which must be started and finished and during which no other communication should take place. Within the API a global matrix can be filled columnwise or blockwise. On CRAY T3E large matrices must be filled by larger blocks as there is a limit in the number of MPI messages which can be open simultaneously, and columnwise filling of global matrices causes too many open messages.

To start PLAPACK and the API and fill a global matrix, e.g. column block wise, the user has to do the following after initializing MPI (again N=M=K):

```
...
/* Create a 2D-Communicator */
PLA_Comm_1D_to_2D(MPI_COMM_WORLD, mp, np, &comm);
/* Create an object distribution template */
PLA_Temp_create( nb_distr, ist0_1, &templ );
/* Create the global matrices */
PLA_Matrix_create( datatype, N, N, templ,
                  PLA_ALIGN_FIRST, PLA_ALIGN_FIRST, &A );
...
/* Create a global scalar */
PLA_Mscalar_create( MPI_DOUBLE, PLA_ALL_ROWS,
                    PLA_ALL_COLS, 1, 1, templ, &alpha );
...
/* Initialize the matrices to equal zero */
  PLA_Obj_set_to_zero ( A );
...
/* Enter Application Interface mode */
PLA_API_begin();
/* Open object A, ... for read/write  */
PLA_Obj_API_open(A);
```

```
...
/* Create a work buffer for computing one column block */
/* of the global matrix A */
/* locA is a local array */
locA = pla_calloc(N*fill_blk_size, type_size);
/* Column blocks are computed by processors in a round-robin fashion */
for (j=me*fill_blk_size;j< N; j+=nprocs*fill_blk_size) {
   int jb, jj;
   jb = min( fill_blk_size, N-j );
   /* Fill column block j of width fill_blk_size */
   for (jj=0; jj<jb; jj++) {
     for (i=0; i < N; i++) {
        ((double *)locA)[jj*N+i] = function of (j+jj,i) ;
     }
   }
   /* Add the column block locA containing jb columns */
   /* to the global matrix A at the location starting with*/
   /* global index (ist0_1+0,ist0_1+j) */
   PLA_API_axpy_matrix_to_global( N, jb, &d_one, locA,
                                  N, A, 0,j );
   /* synchronization after filling in a block of A */
   PLA_Obj_API_sync(A);
}
/* Close the objects */
PLA_Obj_API_close(A);
...
/* Free the workspace */
pla_free( locA );
/* leave Application Interface mode */
PLA_API_end();
```

3.3 Global Arrays

The usage of Global Arrays is described in Th. Steinke's[9] article in these proceedings.

4 Performance

All performance measurements for the matrix-multiplication routines and the routines for the solution of linear systems with LU-decomposition from ScaLAPACK/PBLAS, PLAPACK and Global Arrays and for the solution of the full symmetric eigenvalue problem with ScaLAPACK and Global Arrays (PeIGS) were done on a 256-node CRAY T3E-900 with 128 MB RAM on each node. Some diagrams with performance results are shown in the appendix.

Execution times were measured for various block sizes and grid shapes and in the diagrams collected in the appendix always the shortest time for each matrix

size and processor number is shown.

Additionally we looked at the routines with the performance analysis tool PAT[10]. We used this tool to find out which part of the execution time was spent with communication and in different BLAS routines and to get operation counts per node to see how well the load was balanced.

4.1 Matrix-multiplication and LU-decomposition

We measured execution times for the multiplication $C = 2AB + 3C$ with A, B, and C square matrices of size n and for the LU-decomposition of a square matrix A of size n with the solution of the resulting triangular system with n right-hand-sides. The values of n were $n = 1200, \ldots, 6000$ on 12 nodes, $n = 6000, \ldots, 12500$ on 64 nodes and $n = 6000$ on 10 to 50 nodes.

Global Arrays uses an interface to ScaLAPACK for the solution of a linear system with LU-decomposition. The block size for matrix distribution is 64 for all blockings. This is fixed as a parameter in the interface routine.

For matrix-multiplication Global Arrays contains a routine which uses a blocked version of the usual nested loops, distributes matrix blocks to the processors and calls SGEMM[b] on each node. As this leads to high communication, we modified the ScaLAPACK interface for LU-decomposition to one for matrix-multiplication. This resulted in much better performance for all problems we measured.

4.2 Solution of the Full Symmetric Eigenvalue Problem

For the solution of the full symmetric eigenproblem we measured execution times for the computation of all eigenvalues and eigenvectors of a real full symmetric matrix of size n. The times were measured on 4, 8, 16, 25, 32, 36, and 64 nodes and problem sizes varied from $n = 400$ on four nodes and $n = 800$ on more than four nodes to the maximum n possible on that number of nodes.

A detailed study of the performance of the dense symmetric eigensolvers from ScaLAPACK and PeIGS called by Global Arrays can be found in an internal report of Research Centre Jülich[11].

ScaLAPACK contains two driver routines for the solution of the full symmetric eigenproblem, PSSYEVX, the so-called expert-driver, and PSSYEV, the simple driver. Both compute eigenvalues and optionally eigenvectors by a three-step-algorithm: Reduction of the full matrix to tridiagonal form via Householder transformations, computation of the eigenvalues and (optionally) the eigenvectors of the tridiagonal matrix and back transformation of the eigenvectors to those of the original matrix. Global Arrays' routine GA_DIAG_STD calls PDSPEV from PeIGS and uses the same three steps to compute all eigenvalues and eigenvectors of a real full symmetric matrix.

PSSYEVX and GA_DIAG_STD use parallel bisection and inverse iteration for the computation of the eigenvalues and eigenvectors of the tridiagonal matrix whereas in PSSYEV the eigenvalues of the tridiagonal matrix are computed redundantly (and sequentially) on all nodes via a modified QR-algorithm and only

[b] BLAS 3 routine for matrix-matrix-multiplication, contained in libsci

the computation of the eigenvectors is done in parallel.

PSSYEV only allows to compute all eigenvalues of the matrix and optionally all eigenvectors whereas in PSSYEVX the user can choose a range of eigenvalues to be computed with or without the corresponding eigenvectors. GA_DIAG_STD always computes all eigenvalues and all eigenvectors.

If there are clusters of eigenvalues inverse iteration does not guarantee orthogonality of the corresponding eigenvectors and therefore they have to be reorthogonalized if orthogonal eigenvectors are required. This is done on one single processor for one cluster in PSSYEVX.

If there is one very large cluster of eigenvalues (more than 2000) there is not enough memory (128 MB RAM) for the reorthogonalization of the eigenvectors of this cluster on one node. There is an additional parameter ORFAC in the calling sequence of PSSYEVX which does not appear in SSYEVX, the corresponding LAPACK routine. If ORFAC is set to zero, no reorthogonalization is done and execution times of PSSYEVX are the same whether eigenvalues are clustered or not. Eigenvectors are no longer orthogonal to machine precision. However, the eigenvectors still are nearly orthogonal to an accuracy which might be sufficient in many cases.

In GA_DIAG_STD this problem is solved by a parallel version of the reorthogonalization. It is said in the Users' Guide that it does not guarantee to always deliver orthogonal eigenvectors but in our study we didn't find a case where it didn't work.

The modified QR-algorithm of PSSYEV guarantees orthogonal eigenvectors even for large clusters of eigenvalues and it is even a littlebit faster with one large cluster than without clusters. On the other hand, it needs about twice as many operations per node as PSSYEVX if no eigenvectors have to be reorthogonalized.

4.3 Factors that Influence Performance

There are many factors that affect performance on an MPP system. The user can influence some of them, but others are only influenced by choosing between the libraries.

4.3.1 Usage of BLAS Routines

All tested library routines use BLAS routines for single node computations, so vendor optimized BLAS routines, on CRAY T3E those from libsci, are an important factor for performance. Due to the small level 1 cache on T3E and BLAS 1 routines becoming very slow when data are not in level 1 cache, for all but very small problems performance of BLAS 1 routines is very poor. BLAS 2 routines still can not deliver high performance, so it is preferable to use BLAS 3 routines because cache reuse is possible here.

For matrix-multiplication and the solution of a linear system with LU-decomposition, it is no problem to use BLAS 3 routines. As Global Arrays utilizes an interface to ScaLAPACK for solving linear systems via LU-decomposition there is almost no difference in BLAS 3 usage between both libraries.

Although PAT shows higher communication overhead and lower BLAS 3 usage for PLAPACK than for ScaLAPACK, PLAPACK is faster for large problems.

Therefore, we think that PAT results seem to be not significant in the case of linear system solution. This is probably due to the fact that we could not find out how much of the communication overhead indicated for PLAPACK was due to the filling of the global matrix in the beginning and how much was due to the tested routine.

The usage of BLAS 3 routines plays an important role when comparing the routines for the solution of the symmetric eigenvalue problem. From Table 1 it can be seen that GA_DIAG_STD from Global Arrays, which calls PDSPEV from PeIGS, is based on BLAS 1 routines SDOT and SAXPY whereas PSSYEV and PSSYEVX from ScaLAPACK call the BLAS 3 routine SGEMM and the BLAS 2 routine SGEMV whenever possible. As a result if there is no large cluster of eigenvalues whose eigenvectors have to be reorthogonalized PSSYEVX is much faster than GA_DIAG_STD and reaches a much higher MFLOPS rate per node (see Table 2), although they both use the same algorithm.

Table 1. Percentage of time spent in different BLAS routines. The ranges in percentage arise from different numbers of nodes. For large problems $\geq$ means that for larger problems the percentage is still higher.

no clusters	percentage of time spent in	small problem 200×200 elements per node	large problem $\geq 1000 \times 1000$ elements per node
PSSYEVX	BLAS 3 SGEMM	6 - 8%	$\geq$ 30 %
	BLAS 2 SGEMV	7 - 8 %	$\approx$ 27 %
	BLAS 1	-	-
PSSYEV	BLAS 3 SGEMM	3 - 4 %	$\geq$ 9 %
	BLAS 2 SGEMV	2 - 5 %	7 - 9 %
	BLAS 1 SROT	26-33 %	$\geq$ 56 %
GA_DIAG_STD	BLAS 2, 3	-	-
	BLAS 1 SDOT	28-41 %	39-58 %
	BLAS 1 SAXPY	7 -8 %	19-27 %

If there is a large cluster of eigenvalues the usage of BLAS 3 and BLAS 2 in PSSYEV even is a littlebit higher than in the case of non-clustered eigenvalues whereas BLAS 1 usage in GA_DIAG_STD remains almost the same. This leads to higher MFLOPS rates for PSSYEV than for GA_DIAG_STD and consequently to shorter execution times for non-clustered as well as for clustered eigenvalues.

4.3.2 Load Balance and Communication Overhead

The communication overhead is another important factor for MPP performance. Problems must not be too small for a larger number of nodes because more nodes usually mean more communication and less computation per node.

The algorithm choosen also influences communication overhead as can be seen with the matrix-multiplication routine contained in Global Arrays. For a 1000×1000-matrix on 4 nodes the original routine spends only 60 % in SGEMM and 24 % in communication whereas with the ScaLAPACK interface 77 % of the time is spent

in SGEMM and 11 % in communication. Also load balance is worse in the original routine, three of the four nodes perform about 1800 million floating point operations (MFLOP) and the fourth one only about 750 MFLOP,wheras with ScaLAPACK all processors perform about 1600 MFLOP.

Load imbalance leads to a high communication overhead as a lot of time is spent in waits for other processors to finish computation and send data needed to continue. From Table 2 it can be seen that the ScaLAPACK routines have better balanced operation counts than GA_DIAG_STD in the case with no clusters. For small problems the node with most operations has about 8 to 23 % more operations in PSSYEVX, about 5 to 42 % more in PSSYEV and about 50 to 68 % more operations in GA_DIAG_STD than the node with least operations. With large matrices this becomes more extreme. Whereas in both ScaLAPACK routines the difference is less than 10 % of the operation count of the node with least work, in GA_DIAG_STD the node with most operations has up to 70 % more operations to do than the one with least operations.

Table 2. Millions of floating point operations and MFLOPS per node, equally spread eigenvalues. The operation counts are the lowest and the highest value per node as delivered by PAT, the MFLOPS are computed by the times measured and these operation counts. Only the highest MFLOPS/node rate is shown. On the other nodes MFLOPS rates are lower mainly because of waits.

Million operations per node (MFLOPS per node)	no clusters	small problem 200×200 elements per node	large problem 1000×1000 elements per node
PSSYEVX	4 nodes	123-138 (105)	12300-12600 (270)
	32 nodes	329-365 (90)	33000-34100 (270)
	64 nodes	402-495 (80)	46100-48200 (270)
PSSYEV	4 nodes	263-276 (115)	28400-28600 (215)
	32 nodes	684-964 (100)	75500-81300 (195)
	64 nodes	994-1410 (95)	112000-122000 (185)
GA_DIAG_STD	4 nodes	124-186 (100)	14500-22000 (115)
	32 nodes	349-585 (75)	37600-64800 (80)
	64 nodes	496-768 (60)	54500-92900 (75)

In the case of one large cluster of eigenvalues load balance remains almost the same for PSSYEV. For GA_DIAG_STD it becomes more imbalanced as reorthogonalization plays an important role.

The most extreme example for load imbalance is the reorthogonalization of eigenvectors belonging to a large cluster of eigenvalues which is done sequentially on one single node in PSSYEVX. There it can be seen that with 64 nodes and a problem size of $n = 1600$ and a cluster of 1333 eigenvalues about 94 % of the execution time summed up over all nodes is spent in communication/wait. 62 of the 64 nodes only have to execute about 400-450 MFLOP, one node about 750 MFLOP (orthogonalization of the eigenvectors belonging to one smaller cluster), and one node has to perform about 22100 MFLOP.

4.3.3 Block Sizes and Grid Shapes

ScaLAPACK as well as PLAPACK allow the user to choose block sizes for distribution of vectors and matrices. This size can influence load balance and communication overhead. Small blocks lead to better load balance but to higher communication.

Block sizes have more influence on the performance of routines for solving linear systems than on the performance of routines for the solution of the full symmetric eigenvalue problem.

In ScaLAPACK the system matrix for LU-decomposition has to be distributed into square blocks, i.e. $MB = NB$, but the matrix of the right-hand-sides may be distributed to rectangular blocks with the columns distributed like the system matrix and the rows to blocks of size $NBRHS$.

We found out that for problems with small matrix parts per node ($n = 6000$ on 40 nodes) small block sizes for the system matrix (here $NB = 32$) were best whereas for problems with large matrix parts per node ($n = 6000$ on 12 nodes) larger blocks (here $NB = 64$) were better. For the blocking of the right-hand-sides always a large block size (here $NBRHS = 64$) was best. Powers of two often were slightly better than other numbers of block sizes even if the matrix size wasn't a power of two.

PLAPACK allows to choose only one block size, the one for the distribution of the template vector. Here we could find that as in ScaLAPACK problems with small parts per node perform better with small block sizes ($NB = 32$) and systems with large parts per node with large ($NB = 64$) ones.

For the solution of the symmetric eigenvalue problem with ScaLAPACK routines the system matrix also has to be distributed to square blocks. Here smaller blocks, $NB = 16$ or $NB = 20$ gave best results. For some of the largest problems tested $NB = 32$ delivered the fastest result.

Usually the differences were rather small, but there is one case where the difference is significant. We found out that for block sizes of $NB = 16$ or $NB = 32$ PSSYEVX needs up to twice the time as with a block size of $NB = 20$ if one of the nodes or all the nodes have a local matrix of size 1024×1024 (e.g. 2×2 nodes, $n = 2048, NB = 16$: 80 sec, $NB = 20$: 52 sec execution time; 6×6 nodes, $n = 6000, NB = 32$: 282 sec, $NB = 20$: 133 sec). For PSSYEV the difference is almost the same. But as the execution times are higher time increases only by 50 %. This is due to a performance problem of SGEMM from libsci with the first matrix not transposed and the second one transposed, which is called in the back transformation of the eigenvectors. Called with random matrices the time for SGEMM in the above situation is 4.6 sec for $n = 1000$, 54.8 sec for $n = 1024$ and 6.1 sec for $n = 1050$, hence it takes almost 9 times as long to multiply two 1024×1024 matrices, the second one transposed, than to multiply two 1050×1050 matrices. Therefore it is better not to use powers of two as block sizes even though sometimes the performance is better with those block sizes.

As mentioned in section 2, all libraries presented here distribute matrices to a two-dimensional processor grid. ScaLAPACK and PLAPACK allow the user to explicitely choose the shape of this grid whereas Global Arrays only allows to

determine the minimum number of rows or columns which must be in one block.

For the solution of the symmetric eigenvalue problem with ScaLAPACK the shape of the grid usually does not influence performance very much. If the number of nodes is a square, a square grid achieves highest performance. On rectangular grids, e.g. 8 nodes, sometimes a 2×4 grid and sometimes a 4×2 grid delivers slightly better performance results.

For the solution of linear systems via LU-decomposition grid shapes have more influence on performance than for the solution of the symmetric eigenvalue problem. The time for the solution of the triangular system with n right-hand-sides after LU-decomposition of the matrix with ScaLAPACK is very sensitive to grid shapes. E.g. we found out that in the case of 26 nodes for $n = 6000$, $NB = 32$, and $NBRHS = 64$ on a 2×13-grid the LU-decomposition time was about 19.6 sec and the solution time 139.2 sec whereas on a 13×2-grid the LU-decomposition time was about 18.8 sec and the solution time was only about 61.1 sec. This means that the total time (LU-decomposition + solution) was only half as high on a 13×2-grid than on a 2×13-grid.

4.3.4 Memory Requirements

As mentioned in section 4.3.2 problem sizes per node have to be large to get high performance. High additional memory requirements can therefore cause low performance because the problem size per node can't be made large enough.

Due to the necessity to hold at least a small part of the global data as a local copy when filling the global matrix, we were not able to solve as large linear systems with Global Arrays or PLAPACK as with ScaLAPACK.

On 64 nodes, the largest problem we could solve with the Global Arrays interface to ScaLAPACK was $n = 12288$, with PLAPACK $n = 12800$, and with ScaLAPACK $n = 19000$. Performance of ScaLAPACK was still increasing from less than 300 MFLOPS per node for $n = 12000$ to 350 MFLOPS per node for $n \geq 17000$.

For the symmetric eigenvalue problem we did not see a large difference in memory usage between ScaLAPACK and Global Arrays. This is because PSSYEVX needs additional space for reorthogonalization of eigenvectors as matrices larger than $n = 1000$ tend to have at least one very large cluster due to a nonscalable definition of clusters to remain consistent with LAPACK (see ScaLAPACK Users' Guide). PSSYEV needs additional space for the solution of the tridiagonal eigenvalue problem. For large problems the tridiagonal matrix must be stored on each node no matter how many nodes are used.

4.4 Performance Results

In the appendix we show some diagrams with results of performance measurements.

MFLOPS shown in the figures for matrix-multiplication and LU-decomposition were not taken from the MFLOP counts per node delivered by PAT but were based on the number of floating point operations necessary to solve the problem ($2n^3$ for multiplication of two $n \times n$ matrices and $\frac{8}{3}n^3 - \frac{1}{2}n^2$ for LU-decomposition and solution of an $n \times n$ linear system with n right-hand-sides) divided by the number of nodes and divided by the time the slowest node needed to complete computation.

The MFLOP counts shown by PAT are of course higher than the ones computed because there is always some parallelization overhead and PAT counts operations of the whole program including initialization and collection of results.

Figure 3 shows the results of matrix-matrix-multiplication routines. It can be seen that on 12 processors PLAPACK reaches stable performance of more than 500 MFLOPS/node. ScaLAPACK's PBLAS performance is in the same range and perhaps there could be some better block sizes for $n = 1800, 3000, 4200, 5400$ to get the same performance as with the other sizes. The original Global Arrays routine never reaches 500 MFLOPS/node, thus the ScaLAPACK interface is really necessary for performance.

From figures 4 and 5 it can be seen that for 12 nodes $n = 6000$ is large enough to get high MFLOPS rate of about 370 MFLOPS/node for the solution of linear systems with n right-hand-sides via LU-decomposition with PLAPACK and ScaLAPACK. On 64 nodes, however, any routine delivers poor performance of less than 250 MFLOPS/node with $n = 6000$. On 64 nodes the performance differences become higher. The PLAPACK routine already reaches 250 MFLOPS/node at problem sizes of less than $n = 7000$, whereas this was the highest performance we got with Global Arrays calling ScaLAPACK. ScaLAPACK performance is still increasing with problem sizes $n \geq 12000$ and reaches 350 MFLOS for $n \geq 17000$ as mentioned in section 4.3.4. For both matrix-multiplication and LU-decomposition with solution of the resulting triangular system PLAPACK delivers highest MFLOPS rates and therefore the shortest execution times.

Figures 6 and 7 show execution times for the computation of all eigenvalues and all eigenvectors of a real symmetric matrix of size $n = 2000, \ldots, 2500$. Figure 6 shows the times in the case where the eigenvalues are equally spread and reorthogonalization is not necessary. Figure 7 shows execution times for a matrix with one large cluster of $n - 267$ eigenvalues. In PSSYEVX and GA_DIAG_STD the eigenvectors belonging to this cluster are reorthogonalized. It can be seen that for equally spread eigenvalues PSSYEVX on 4 nodes is as fast as PSSYEV on 16 nodes. One reason, of course is, that PSSYEV needs twice as many operations as PSSYEVX. The other reason is, that the sequential QR-algorithm within PSSYEV uses a lot of BLAS 1 routines and consequently reaches a lower MFLOPS rate than PSSYEVX.

Although the number of operations of GA_DIAG_STD is almost the same as the number of operations of PSSYEVX (see Table 2), performance is much slower. It is even slower than the performance of PSSYEV with twice as many operations. One reason for this behaviour is the fact that it is completely based on BLAS 1 routines and therefore performance is reduced by cache misses. Thus the MFLOPS rates per node reached with GA_DIAG_STD are significantly lower than the ones reached by PSSYEV. Another reason is the poorer load balance of GA_DIAG_STD, which also may be seen from Table 2.

If there is one large cluster of eigenvalues and the eigenvectors have to be reorthogonalized the situation changes dramatically. Now on four nodes the largest problem that could be solved with PSSYEVX was $n = 2000$. The execution times for PSSYEVX are almost independent of the number of nodes.

It can be seen that the execution times for PSSYEV are slightly lower in the

case of one large cluster of eigenvalues than in the case of equally spread eigenvalues. The execution times of GA_DIAG_STD on the other hand become higher as eigenvectors are reorthogonalized. Consequently the difference between PSSYEV and GA_DIAG_STD becomes larger than in the case without a cluster.

For the solution of the symmetric eigenvalue problem there is always one ScaLAPACK routine with highest performance: if eigenvalues are not clustered this is PSSYEVX, for one large cluster of eigenvalues whose eigenvectors have to be reorthogonalized PSSYEV is the fastest routine.

5 Conclusions

ScaLAPACK offers very good performance compared to the other libraries and a broad range of black box solvers but at the expense of a little more complicated user interface. Programmers willing to apply ScaLAPACK routines should become familiar with the data distribution used in ScaLAPACK and adapt their program to this distribution from the start. This will result in good performance and low memory usage.

PLAPACK achieves highest performance on those routines available in that library but these are only a few and the lack of a solver for the symmetric eigenvalue problem will prevent most people having to solve eigenvalue problems from using it. "Using PLAPACK"[3] explains how to use PLAPACK for writing linear algebra routines based on the PLAPACK distributed objects but we think it needs an experienced user to write an eigensolver based on PLAPACK.

Global Arrays mainly offers an infrastructure to treat global objects transparently. If routines from other libraries are to be used this costs some performance due to redistribution of data. There is only one interface routine to ScaLAPACK within Global Arrays, the one for LU decomposition and the solution of the resulting triangular system. This routine must be modified if other ScaLAPACK or PBLAS routines like matrix-matrix-multiplication shall be used. It seems to be much easier to write an interface to use PSSYEV(X) from Global Arrays than to write a new eigensolver for PLAPACK.

All libraries can help developers of new application programs or application packages to take advantage of work already done. They are not meant for those who only want to use software on the application level.

References

1. *HPC-Netlib Homepage*, http://www.nhse.org/hpc-netlib/
2. L.S. Blackford, J. Choi, A. Cleary et al., *ScaLAPACK Users' Guide*, SIAM, Philadelphia (1997).
3. R.A. van de Geijn, *Using PLAPACK: Parallel Linear Algebra Package*, The MIT Press, Cambridge, Massachusetts (1997).
4. *Global Arrays User Guide*, http://www.emsl.pnl.gov:2080/docs/global/ga.html
5. E. Anderson, Z. Bai, C. Bischof et al.: *LAPACK Users' Guide, Second Edition*, SIAM, Philadelphia (1995).

6. J.J. Dongarra and R.C. Whaley, *A User's Guide to the BLACS v1.1*, LAPACK Working Note 94, (1997), http://www.netlib.org/lapack/lawns/lawn94.ps
7. Silicon Graphics, CRAY Research: *Scientific Libraries Reference Manual, Vols. 1-2*, CRAY Research, Inc. (1997).
8. D. Elwood, G. Fann, R. Littlefield, *Parallel Eigensystem Solver PeIGS Version 2.1, rev. 0.0*, Pacific Northwest National Laboratory (July 28, 1995), http://www.emsl.pnl.gov:2080/docs/nwchem/highlights/peigs/peigs.html
9. Th. Steinke, *Tools for parallel quantum chemistry software*, Modern Methods and Algorithms of Quantum Chemistry, Proceedings, Jülich, Germany (February 2000).
10. J. Galarowicz, B. Mohr, *Analyzing Message Passing Programs on the Cray T3E with PAT and VAMPIR*, Forschungszentrum Jülich GmbH, Zentralinstitut für Angewandte Mathematik, Interner Bericht, FZJ-ZAM-IB-9809, Mai 1998.
11. I. Gutheil and R. Zimmermann, *Performance of Software for the Full Symmetric Eigenproblem on CRAY T3E and T90 Systems*, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Interner Bericht, to appear.

Appendix

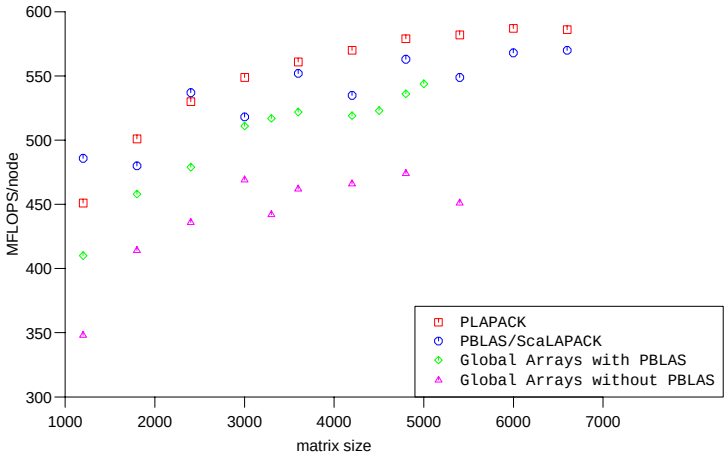

Figure 3. Matrix-matrix-multiplication $C = 2AB + 3C$, A, B, C square matrices, using different library routines on 12 nodes.

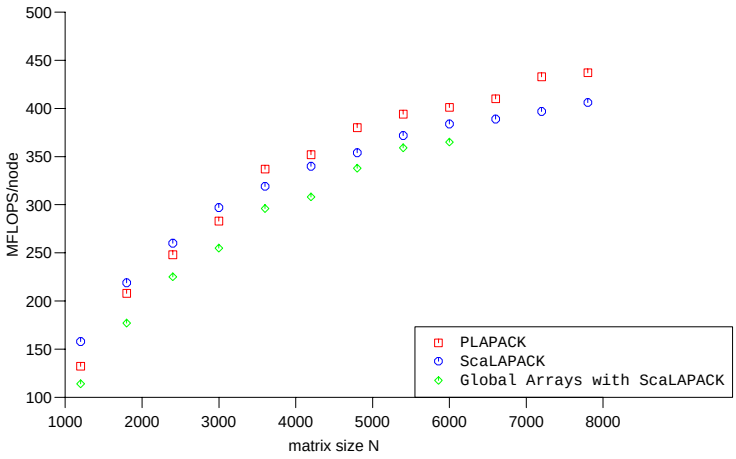

Figure 4. Solving a linear system with N right-hand-sides by means of LU-decomposition using different library routines on 12 nodes.

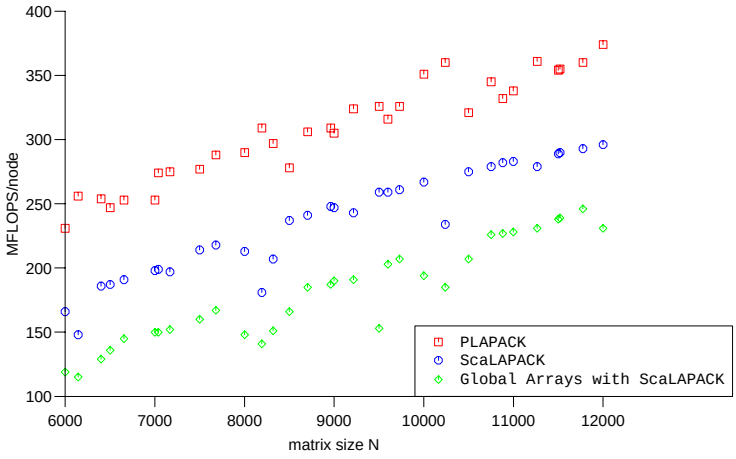

Figure 5. Solving a linear system with N right-hand-sides by means of LU-decomposition using different routines on 64 nodes.

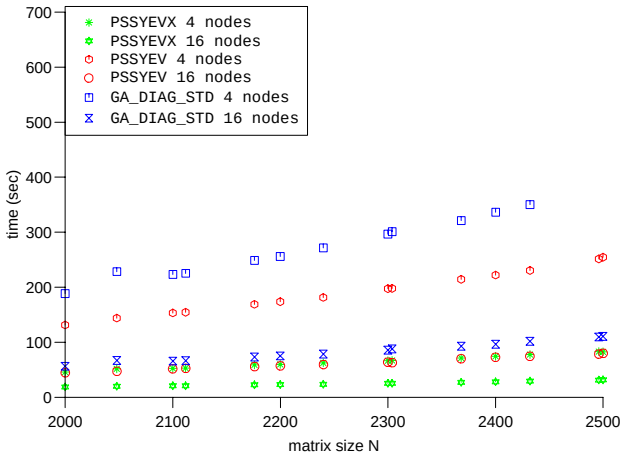

Figure 6. Computation of all eigenvalues and eigenvectors of an $N \times N$-matrix (no clusters of eigenvalues) using different library routines.

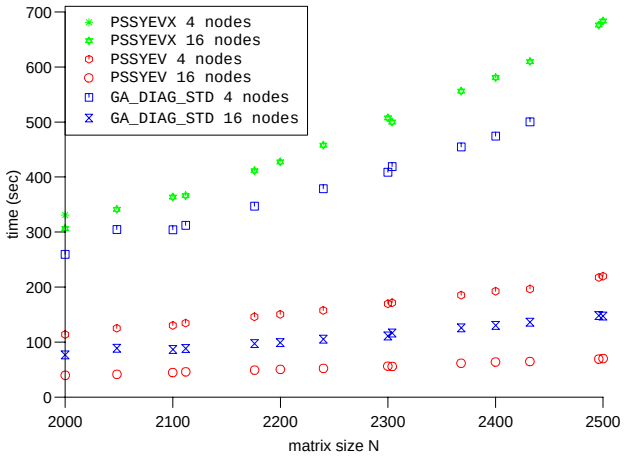

Figure 7. Computation of all eigenvalues and eigenvectors of an $N \times N$-matrix with one large cluster of eigenvalues using different library routines.