Implementing Iterative Solvers for Irregular Sparse Matrix Problems in High Performance Fortran

E. de Sturler[1] and D. Loher[1]

Swiss Center for Scientific Computing (SCSC-ETHZ), Swiss Federal Institute of Technology Zurich, ETH Zentrum (RZ F-11), CH-8092, Zurich, Switzerland, email: sturler@scsc.ethz.ch, phone: +41-1-632 5566, fax: +41-1-632 1104

Abstract. Writing efficient iterative solvers for irregular, sparse matrices in HPF is hard. The locality in the computations is unclear, and for efficiency we use storage schemes that obscure any structure in the matrix. Moreover, the limited capabilities of HPF to distribute and align data structures make it hard to implement the desired distributions, or to indicate these such that the compiler recognizes the efficient implementation.
We propose techniques to handle these problems. We combine strategies that have become popular in message-passing parallel programming, like mesh partitioning and splitting the matrix in local submatrices, with the functionality of HPF and HPF compilers, like the implicit handling of communication and distribution. The implementation of these techniques in HPF is not trivial, and we describe in detail how we propose to solve the problems. Our results demonstrate that efficient implementations are possible. We indicate how some of the 'approved extensions' of HPF-2.0 can be used, but they do not solve all problems. For comparison we show results for regular, sparse matrices.

Keywords: High Performance Fortran, Irregular Sparse Matrices, Iterative Solvers.

1 Introduction

For large, sparse linear systems we often use iterative methods because of their efficiency in both memory requirements and work. On parallel computers, iterative methods have the additional advantage that they do not change the structure of the problem. Iterative methods use four major kernels: matrix-vector product, preconditioner, inner product (ddot), and vector update (daxpy). The choice of a preconditioner is very important for the efficient solution of a linear system, but we will not discuss preconditioning here because it is often problem-dependent. However, very effective parallel preconditioners have been derived that have essentially the same communication requirements as the matrix-vector product [2, 3, 6]. For regular sparse, linear systems, like those derived from regular grids, using High Performance Fortran (HPF) for iterative solvers is straightforward.

However, for irregular sparse matrices the efficient implementation of solvers in HPF becomes much harder.

First, the locality in the computations (a good partitioning) is unclear. Second, for efficiency we often use storage schemes that obscure even the simplest structure in the matrix (like rows and columns). Third, the limited capabilities of HPF to distribute data structures make it hard to implement the desired distribution. Fourth, data structures often have very different sizes and shapes, and matching the distributions for efficient implementation (locality) is a problem. Fifth, after implementing the distributions, we still must write the program in such a way that the compiler recognizes the efficient implementation, and leaves out unnecessary communication, synchronization, etc.

We propose techniques for handling these problems, and our results demonstrate that efficient implementations are possible. In [4, 5] we showed that, unless special implementations are chosen, on large parallel computers the global communication in the inner products dominates the parallel performance of iterative solvers. This is an architectural feature; it is independent of whether we use HPF or, say, MPI. The cost of an inner product is about the same in both implementations. Clearly, if we can find implementations for the sparse matrix-vector product for irregular matrices that are more efficient than the inner product, the sparse matrix-vector product will not be the bottleneck. We will show that this is indeed possible, even for relatively small problems. For comparison, we show results for regular, sparse matrices. The generalized block distribution GEN_BLOCK in the 'approved extensions'[1] of the HPF Language Specification version 2.0 (HPF-2.0) [7] solves the distribution problems (problem three in the discussion above), but not the other problems. We will discuss this at the end of the paper.

All our experiments are carried out using the Portland Group (PGI) HPF compiler (version 2.1) on the Intel Paragon at the Swiss Federal Institute of Technology (ETH Zurich).

In the next section we discuss the parallel performance of iterative methods for regular, sparse matrices for comparison. For our experiments we use the GMRES method [9] (see Fig. 1), one of the most widely used iterative methods. In Section 3, we outline how we address the problems mentioned above, and we discuss the performance results. In Section 4, we indicate in how far the approved extensions of HPF-2.0 improve the situation, and what is still needed. In the last section we provide some future directions.

2 Regular Sparse Matrix Problems

For regular problems, such as k-diagonal matrices resulting from discretizations over regular grids, the parallelization with HPF is straightforward. Our test problem comes from a convection-diffusion problem, discretized on a regular, two-dimensional grid (501×501), with Dirichlet boundary conditions. The matrix

[1] It will probably take more than a year before such extensions become available.

```
GMRES(m):

start:
  $x_0$= initial guess; $r_0 = b - Ax_0$;
  $v_1 = r_0/\|r_0\|_2$;

iterate:
  for $j = 1, 2, \ldots, m$ do
    $\hat{v}_{j+1} = Av_j$;
    for $i = 1, 2, \ldots, j$ do
      $h_{i,j} = (\hat{v}_{j+1}, v_i)$;
      $\hat{v}_{j+1} = \hat{v}_{j+1} - h_{i,j} v_i$;
    end;
    $h_{j+1,j} = \|\hat{v}_{j+1}\|_2$;
    $v_{j+1} = \hat{v}_{j+1}/h_{j+1,j}$;
  end
form the approximate solution:
  $y_m = \arg\min_{y \in I\!R} \left\| \, \|r_0\|_2 e_1 - \bar{H}_m y \right\|_2$;
  $x_m = x_0 + V_m y_m$;

restart:
  $r_m = b - Ax_m$;
  if $\|r_m\|_2 < tol$ then
    stop
  else
    $x_0 = x_m$; $v_1 = r_m/\|r_m\|_2$;
    goto iterate
  end
```

Fig. 1. The GMRES(m) algorithm.

and vectors are stored by grid point (reflecting the underlying two-dimensional structure), which leads to an efficient implementation of the communication in the matrix-vector product.

Tables 1 and 2 show that for one to sixteen processors the efficiency for the matrix-vector product is eighty percent or higher, and this is not significantly lower than the efficiency for the other routines. More important, these tables show that for larger numbers of processors (32 and more), the speed-up for GMRES(40) is dominated by the inner products (ddot), not by the matrix-vector product. Especially, notice the similarity between the speed-up figures for GMRES(40) and the inner product (ddot) for eight processors or more. Note that increasing the restart cycle (here 40) of GMRES leads to a quadratic increase in the number of inner products compared with a linear increase in the number of matrix-vector products. Hence, for larger numbers of processors a higher efficiency for the matrix-vector product than for the inner products has no influence

Table 1. Runtimes in seconds for sequential and parallel GMRES(40), matrix-vector product (matvec), inner product (ddot), and vector update (daxpy). The large run-times between brackets for one and two processors are caused by swapping.

#proc	1	2	4	8	16	32	64
gmres	(2.75E+03)	(1.18E+03)	9.17E+01	4.79E+01	2.68E+01	1.62E+01	1.28E+01
matvec	1.29E+00	6.44E-01	3.79E-01	1.96E-01	1.02E-01	5.50E-02	3.61E-02
ddot	1.91E-01	9.60E-02	4.95E-02	2.56E-02	1.46E-02	8.57E-03	6.72E-03
daxpy	1.12E-01	5.29E-02	2.66E-02	1.34E-02	7.02E-03	3.62E-03	1.88E-03

on the performance of the algorithm as a whole.

The large difference in the run-time of one GMRES(40) iteration for the sequential program and the parallel program on two processors compared with the parallel program on four and more processors is caused by swapping in the former case; the data does not fit in the memory of two processors, but it fits in the memory of four processors. This, of course, makes the timings incomparable, and therefore we normalized the speed-up for four and more processors against the run-time on four processors. So the speed-up on four processors is equal to four. The speed-up values for the separate routines, which have been timed without swapping, show that this is not far from the speed-up that would have been measured on a machine that can run the program on one processor without swapping.

3 Irregular Sparse Matrix Problems

For algorithms like GMRES(m) the only difference between structured and non-structured problems is in the implementation of the matrix-vector product and of the preconditioner. As argued above, for many popular parallel preconditioners the communication requirements and the implementation of the communication resemble those of the distributed matrix-vector product, and hence we will concentrate on the latter.

Table 2. Speed-ups for GMRES(40), the matrix-vector product (matvec), the inner product (ddot), and the vector update (daxpy). For GMRES(40), the speed-ups for four and more processors have been derived from the run-time on four processors.

#proc	1	2	4	8	16	32	64
gmres(40)	(1.00)	(2.33)	4.00	7.66	13.7	22.6	28.7
matvec	1.00	2.00	3.40	6.58	12.7	23.5	35.7
ddot	1.00	1.99	3.86	7.46	13.1	22.3	28.4
daxpy	1.00	2.12	4.21	8.36	16.0	28.4	59.6

3.1 Sparse Matrix Storage Schemes

We will first describe two often-used sparse matrix storage schemes, the so-called compressed sparse column (CSC) and compressed sparse row (CSR) scheme. The CSR scheme uses three arrays to describe the matrix. Let *nnz* be the number of non-zero coefficients in the matrix and let n be the number of rows (and columns) in the matrix.

- *val(1:nnz)* contains the values of the non-zero coefficients of the matrix in row-wise order.
- *col_idx(1:nnz)* contains the column indices for the corresponding elements of *val*: *col_idx(i)* gives the column in which coefficient *val(i)* appears.
- *row_ptr(1:n+1)* contains pointers to the start of each row in the arrays *val* and *col_idx*. The last pointer points one past the last element of *val* and *col_idx*.

We can implement the matrix-vector product $y = Ax$ (in Fortran77) as follows.

```
CSR matrix-vector product:
do row = 1, n
  y(row) = 0.0
  do j = row_ptr(row), row_ptr(row+1)-1
    y(row) = y(row) + val(j)*x(col_idx(j))
  end do
end do
```

The CSC scheme is equivalent to the CSR scheme except that the array *val* contains the non-zero coefficients in column-wise order, and therefore we have an array with column pointers and an array with row indices instead of the other way around. Historically the CSC scheme has received a certain preference because it often leads to superior performance on vector computers. However, on parallel machines this is not the case and the disadvantages (especially for implementation of the preconditioner) dominate. We will assume the CSR scheme in this paper; however, our approach is easily converted for the CSC scheme.

3.2 Distribution of the Matrix

Usually, we prefer a row-wise distribution (partitioning) of the matrix. With an appropriate ordering (see below) this amounts to a domain decomposition, which facilitates several effective preconditioning techniques. Also more generally, row-wise distribution makes preconditioning easier to implement (block-ILU type preconditioning e.g.). However, it is straightforward to adapt our approach to a column-wise distribution.

The distribution of the matrix assigns to each processor a 'local matrix' consisting of the set of rows in the partition assigned to that processor. For an efficient implementation of the matrix-vector product the distribution of the arrays describing the matrix must be as follows. For each partition the non-zero coefficients of the local matrix are stored in the local part of the array *val*, the column indices of the local matrix are stored in the local part of the array *col_idx*, and the pointers to the rows of the local matrix are stored in the local parts of the arrays *start_row* and *end_row*, which replace *row_ptr* (see below). We distribute the vectors in the same way as the matrix: we store the i-th coefficient of a vector on the same processor as the i-th row of the matrix.

We use graph partitioning techniques (currently in a separate off-line phase) on the underlying computational grid or directly on the matrix to find a partitioning of the rows of the matrix such that the distributed, sparse matrix-vector product yields low communication cost and a good load balance. Low communication cost means that the total number of non-local references in the matrix-vector product is minimized. Preferably, also the number of processor-pairs that need to exchange data, which is equal to the number of messages, should be minimized. For the matrix-vector product, load balancing means that the number of non-zero matrix coefficients in each local matrix is about the same. However, for the vector operations load balancing means that the number of unknowns on each processor, which is equal to the number of rows in each local matrix, is about the same. Currently we use the package by Simon and Barnard [8, 1] to compute partitionings. This package allows a trade-off between the two different load balancing requirements in computing the partitioning.

From the output of the graph partitioning routine we know which rows of the matrix should be grouped together in a partition (i.e., on a processor) to form the local matrix. For each partition the local matrix should consist of the local parts of the arrays *val*, *col_idx*, and *row_ptr*. This gives a problem for the array *row_ptr*. Since each pointer serves a double purpose, to indicate the start of a row and the end of the previous row, the distribution of this array means we cannot for all rows have the necessary pointers locally available. Therefore, we replace this array by two new arrays; *start_row(1:n)*, with pointers to the start of each row; and *end_row(1:n)*, with pointers to the end of each row. Now we must implement the distribution of the arrays describing the matrix. To achieve this we will use a block-wise distribution and renumber the rows and columns explicitly. However, renumbering by itself is not enough. In general, we do not have the same number of rows in each partition, and we do not have the same number of non-zero coefficients in each partition. So the regular distribution indicated by the HPF DISTRIBUTE (BLOCK) directive does not give the desired distribution. Moreover, there is no fixed ratio between the number of rows and the number of non-zero coefficients, because the number of non-zero coefficients per row may vary strongly between rows. So, the arrays of row pointers differ in size from the arrays with matrix coefficients and column indices, and the ratio between these sizes differs per partition. In short, we cannot use the HPF ALIGNMENT directive to enforce that the pointers to the rows in a partition are stored on the same processor as the coefficients and column indices of that partition.

We solve the distribution and alignment problems as follows. First we introduce dummy (empty) rows such that each partition has the same number of rows. Then we create dummy coefficients such that each partition has the same number of non-zero coefficients. These coefficients will be outside any row, so they will never be used in computations. The dummy rows can be masked, so also here no additional computation is introduced. Moreover, the mesh partitioning algorithm always generates a partitioning with a good load balance, so that the numbers of additional rows and coefficients are negligible and no overhead in memory results. Reordering the padded matrix and distributing the arrays regularly through the HPF block distribution directive now leads to the desired distributions and alignments. We have all information about the local matrices locally available.

However, one problems remains: the compiler has no way of knowing that the arrays with row pointers are actually 'aligned' with the arrays with column indices and coefficients. So a straightforward implementation of the matrix-vector product leads to a large overhead in unnecessary checks and synchronizations, or worse in unnecessary duplication and communication of data, and even in unnecessary computations. Unless we make additional changes to the sizes of the arrays we still cannot use the HPF alignment directive to align the arrays in their rank-one form, or indicate this alignment to the compiler. We have two ways of solving this problem. The first way is to make the necessary communication of the non-local vector values explicit (by a copy) and then use an HPF_LOCAL routine for the matrix-vector product. This way the local availability of all the data is explicitly given, and we avoid problems with the HPF compiler creating unnecessary overhead. The second way is to reshape the arrays describing the matrix into rank-two arrays, because then we can use the HPF alignment directive to align the arrays and make this locality in the matrix-vector product explicit. We will get the following arrays with a partition index and a local index: *val(part_index,loc_idx)*, *col_idx(part_idx,loc_idx)*, *start_row(part_idx,loc_row)*, *end_row(part_idx,loc_row)*. For the vectors we still have several options. This implementation is certainly the most elegant. However, it leads to further complexities in the actual implementation of the matrix-vector product that we cannot go into here. We will discuss this in a future paper. The HPF_LOCAL version has the additional advantage that the local part of the matrix-vector product resembles the sequential version. We will use the HPF_LOCAL version for our tests below.

Finally, we like to point out that the implementation is actually not as complicated as it may seem. Assuming that we read in the matrix in one of the standard storage schemes for 'sequential' matrices, the restructuring is accomplished in a relatively cheap preprocessing phase before the actual iterative solver.

3.3 Tests and Results

After distributing and reordering the data structures the matrix-vector product is quite efficient, and the scheduler creates an efficient communication scheme. In fact, the scheduler itself becomes the most costly part. In general, this is not important because scheduling needs to be done only once for many iterations, and hence the cost becomes negligible if the schedule is reused. In general, the

PGI compiler does move the computation of the communication schedule out of loops; however, in more complex routines like the GMRES algorithm, apparently, this no longer works (insufficient analysis capability), and the schedule is recomputed unnecessarily for every matrix-vector product. We assume that in the future such features will improve. If this is the case, irregular sparse matrix computations become quite feasible with HPF. In order to show that we can achieve a sufficiently efficient matrix-vector product provided the communication schedule is reused, we use three subroutines provided by L. Meadows of PGI to explicitly reuse the schedule.

In the previous section we showed that for larger numbers of processors the efficiency of the inner product tends to dominate the overall efficiency of the iterative solver. So, if the efficiency of the matrix-vector product is higher than that of the inner product, the efficiency of the matrix-vector product has no influence on the overall performance. Therefore, we will only discuss the results for the matrix-vector product and the inner product (ddot) here. For the purpose of analysis we have split the matrix-vector product in the part that fetches all non-local data (gather) and the actual computation (comp). Unfortunately, large, irregular, sparse test matrices are not so easily available, and hence we can only provide results for a problem that is much smaller than what we used for the regular case. The largest matrices in the Harwell-Boeing test collection are of the order of 35000 unknowns. Our test problem (bcsstk31) has 35588 unknowns and 1181416 nonzero coefficients in the matrix. The run-time for the sequential matrix-vector product is 0.268 s. For the inner product (ddot) the sequential run-time is given in Table 3. The parallel run-time of the matrix-vector product is the sum of the time for 'gather' and for 'comp'. Because of the implementation there is a non-zero run-time for 'gather' on one processor (basically a copy). Of course, this could have been masked, but we feel that this information provides useful insight for the performance on multiple processors. For example, we see that the gather (including communication) scales almost linearly from one to eight processors, and only for more processors the efficiency decreases sharply.

Table 3. Run-times in seconds for the irregular, sparse matrix-vector product (matvec), its gather part (gather) and its computation part (comp), and for the inner product (ddot).

#proc	1	2	4	8	16	32
gather	2.09E-01	1.09E-01	5.67E-02	3.16E-02	2.32E-02	2.26E-02
comp	3.75E-01	1.94E-01	9.66E-02	4.44E-02	2.30E-02	1.26E-02
matvec	5.84E-01	3.03E-01	1.53E-01	7.60E-02	4.62E-02	3.52E-02
ddot	1.29E-02	6.78E-03	4.22E-03	2.75E-03	2.25E-03	2.17E-03

We see that also in this case for larger numbers of processors the efficiency of the inner product drops below the efficiency of the irregular, sparse matrix-vector

Table 4. Speed-up and efficiency for the irregular, sparse matrix-vector product (matvec) and for the inner product (ddot).

#proc		1	2	4	8	16	32
matvec	speed-up	0.459	0.884	1.75	3.53	5.80	7.61
	efficiency (%)	45.9	44.2	43.8	41.1	36.3	23.8
ddot	speed-up	1.00	1.90	3.06	4.69	5.73	5.94
	efficiency (%)	100	95.1	76.4	58.6	35.8	18.6

product. So, for larger numbers of processors the irregular, sparse matrix-vector product will not be a bottleneck. Since the inner products are about as fast in HPF as they are in message-passing code (MPI/PVM), these results also show that a message-passing code cannot be much faster on large numbers of processors. We see on the other hand that for smaller numbers of processors the efficiency of the matrix-vector product is between forty and fifty percent. So, for our current implementation we will not see performance much above that level for irregular problems. Clearly, for small numbers of processors a message-passing code will do better. However, we think that the efficiency we achieve is high enough to be interesting for many applications. Especially since it seems to be fairly constant over a range of numbers of processors. This indicates that we can expect this level of performance on larger numbers of processors for larger problem sizes, because for many irregular, sparse problems the connectivity of the matrix or mesh is independent of the number of unknowns.

Furthermore, several improvements are still possible. We see that the actual computation (comp) is not so efficient; we do not have the same efficiency for the local computations (without communication) as we had for the original sequential program. We have to improve this. Also the gather part of the matrix-vector product seems too expensive. Probably, this is due to the work involved in masking (basically if-statements). We can adapt the implementation to prevent any references to dummy rows and coefficients.

In the new version of our program, we will split the matrix-vector product even further, so that each local matrix-vector product consists of two parts. One part refers only to the unknowns that are locally available, and the other part refers to the unknowns that are not locally available. The non-local references are stored in a data structure that resembles the one for the matrix as a whole. The implementation of the local part of the matrix-vector product can be exactly the same as the sequential version; this should bring a major improvement for the cost of both the computational part and the gather part. The implementation of the non-local part of the matrix-vector product will be the same as for the previously discussed program as a whole. This too should reduce the cost of the matrix-vector product. With these improvements we expect to significantly raise the efficiency and speed-up of the matrix-vector product for (relatively) small numbers of processors.

4 HPF-2.0 and Extensions

The new High Performance Fortran Language Specification version 2.0 [7] (HPF-2.0) includes a separate part on 'Approved Extensions'. These are *advanced features that meet specific needs, but are not likely to be supported in initial compiler implementations.* Given the fact the the standard HPF-2.0 features are expected to be implementable within a year, it is unlikely that the approved extensions will be commonly available soon. So, for portable programs we will have to continue for the moment on the way described in the previous section. However, it is important, to anticipate these new, and important, features. We will show that they solve some of our problems with irregular matrices, but that more is still needed.

The generalized block distribution in its executable form,
HPF REDISTRIBUTE(GEN_BLOCK(*block_sizes*)) allows to compute or read in a desired partitioning and implement this in run time. The array *block_sizes* gives the sizes of the blocks. Using this directive we can distribute the arrays *val* and *col_idx* with appropriate block sizes to give the number of non-zero coefficients in each partition, and the arrays *start_row* and *end_row* with appropriate block sizes to give the number of rows in each partition. Note that a corresponding alignment directive ALIGN(GEN_BLOCK()) does not exist. This would allow us to align the arrays *start_row* and *end_row* with the arrays *val* and *col_idx* so that we have the description of a local matrix locally available. As mentioned before, the arrays with the row pointers differ in size from the arrays with the values and column indices, and there is no regular (linear) relation that matches the row pointers with the values or column indices in the row. The number of non-zero coefficients per row may vary strongly. For our purposes we would like an alignment that matches two compatible generalized block distributions. That is, two block distributions with different sizes but for the same number of partitions. We could map a block of row pointers to the processor that contains the block of coefficients that belong to that row. Such a directive could have the form (RE)ALIGN(GEN_BLOCK(*row_partition,coefficient_partition*)), where *row_partition* and *coefficient_partition* are two rank-one arrays that have the same size.

The REDISTRIBUTE(GEN_BLOCK()) directive can be used to get the desired distributions without padding the arrays with dummy coefficients or dummy pointers. However, since we cannot use an alignment directive, the fact that all information about the local matrix is locally available on each processor is still not clear to the compiler. So, as long as no generalized, block-wise alignment is available, the associated problems have to be handled in the same way as in the previous section.

5 Conclusions

We have outlined an approach to implement irregular, sparse matrix solvers in HPF. Our results clearly show that reasonable speed-ups are attainable, and that the performance of the sparse matrix-vector product does not play a significant role in the scalability. Indeed, for larger numbers of processors (relative to the problem size) the global communication in the inner products dominates the performance, and the efficiency of the inner product drops below the efficiency of the irregular, sparse matrix-vector product.

The current implementation is by no means optimal yet. We mainly concentrated on low communication costs, which we seem to have achieved. Moreover, we probably can improve the communication costs significantly by better partitioning algorithms. Some tests we performed indicated that the current partitionings are not so good. Also the computational cost is too high, and we have indicated several improvements.

Finally, we have indicated how the generalized block distribution in the 'approved extensions' of HPF-2.0 helps with some, but not all, of our problems. We propose a block-wise alignment to improve HPF programs for irregular, sparse matrix algorithms.

Acknowledgements

We would like to thank Larry Meadows (PGI) and Horst Simon (NERSC) for making their codes available to us.

References

1. S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NASA Ames Research Center, Mail Stop T045-1, Moffet Field, CA 94035, USA, 1992.
2. E. De Sturler. *Iterative Methods on Distributed Memory Computers.* PhD thesis, Delft University of Technology, Delft, The Netherlands, October 1994.
3. E. De Sturler. Incomplete Block LU preconditioners on slightly overlapping subdomains for a massively parallel computer. *Applied Numerical Mathematics (IMACS)*, 19:129–146, 1995.
4. E. De Sturler and H. A. Van der Vorst. Communication cost reduction for Krylov methods on parallel computers. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking*, Lecture Notes in Computer Science 797, pages 190–195, Berlin, Heidelberg, Germany, 1994. Springer-Verlag.
5. E. De Sturler and H. A. Van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Applied Numerical Mathematics (IMACS)*, 18:441–459, 1995.
6. F. Nataf, F. Rogier, and E. De Sturler. Domain decomposition methods for fluid dynamics. In A. Sequeira, editor, *Navier-Stokes Equations and Related Nonlinear Problems*, New York, 1995. Plenum Press.
7. High Performance Fortran Forum. High Performance Fortran Language Specification, version 2.0 Rice University, 1997
8. A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11:430–452, 1990.

9. Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.