LAPACK Working Note 137
Installation Guide and Design of the HPF 1.1 interface to ScaLAPACK, SLHPF [1]

L.S. Blackford, J.J. Dongarra
C. A. Papadopoulos, and R. C. Whaley
Department of Computer Science
University of Tennessee
Knoxville, Tennessee 37996-1301

REVISED: RELEASE VERSION 1.1, Sept 1, 1998

Abstract

This working note describes release version 1.1 of the HPF 1.1 compliant interface to ScaLAPACK, SLHPF. Along with a description of the interface, installation instructions, a simple example program, and a testing suite are included. ScaLAPACK is a library of high-performance linear algebra routines for distributed-memory parallel machines or clusters of workstations supporting MPI or PVM. SLHPF allows the user to call the ScaLAPACK library from within an HPF program.

[1] This work was supported by the National Science Foundation Grant No. ASC-9005933; by the Defense Advanced Research Projects Agency under contract DAAH04-95-1-0077, administered by the Army Research Office; by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-84OR21400; and by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615; and by the DoD High Performance Computing Modernization Program ASC Major Shared Resource Center through Programming Environment and Training (PET) under Contract Number DAHC-94-96-C-0005, Nichols Research Corporation, subcontract no. NRC CR-96-0011.

Contents

1 Introduction

SLHPF is an interface layer that allows the user to call ScaLAPACK from an HPF 1.1 program. ScaLAPACK, written in Fortran 77 and C, is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers and networks of workstations supporting PVM or MPI. It is a scalable distributed memory version of LAPACK. ScaLAPACK contains routines for solving systems of linear equations, least squares problems, singular value decomposition, and eigenvalue problems. ScaLAPACK's approach to achieving high efficiency is based on the use of a standard set of Basic Linear Algebra Subprograms (BLAS), which can be optimized for each computational environment. By confining most of the computational work to the BLAS, the subroutines should be transportable and efficient across a wide range of computers. For communication, ScaLAPACK uses the Basic Linear Algebra Communication Subprograms (BLACS) from which a parallel implementation of the BLAS, (PBLAS) are implemented.

The parallel language HPF supports CYCLIC(k) distributions of matrices, and thus is a natural platform from which to call ScaLAPACK. The SLHPF interface is designed to make this approach both easy to use and portable, thus allowing the efficiency and reliability of ScaLAPACK to be realized. HPF compilers are relatively young and the language is large and complex, making it difficult for compiler writers to implement all of the features of the HPF 1.1 standard. In view of this we have chosen a minimum subset of HPF compiler directives necessary to implement the interface and to make the interface as portable as possible. This working note describes how to install and test this release of SLHPF.

2 Revisions Since the First Public Release

Since its first public release SLHPF version 0.2Beta, many of the problems and difficulties with compiler limitations have been addressed. SLHPF release version 1.1 has been rewritten to comply with version 1.1 of the HPF standard. It will not work with a 1.0 compliant HPF compiler. Some of the limitations from the 0.2Beta release that have been addressed are: Redistribution is now supported. Driver routines which accept arguments which may be 2D or 1D arrays (eg. the right hand side vector (or series of vectors) X) now will accept 1D or 2D arrays. A few code segments are still required to write around known compiler bugs; however, include files are still used instead of module files as in the original release. This is because we have had problems with compiler bugs using module files in the past and we wanted as few potential problems with new compilers as possible. We have also increased the functionality by providing an HPF interface to the ScaLAPACK symmetric eigenproblem driver SYEV.

3 File Format

The software for SLHPF is distributed in the form of a gzipped tar file (via anonymous ftp or the World Wide Web) which contains the HPF source for SLHPF, the testing programs, this working note, and a simple example program. The package may be accessed via the World Wide Web through the URL address:

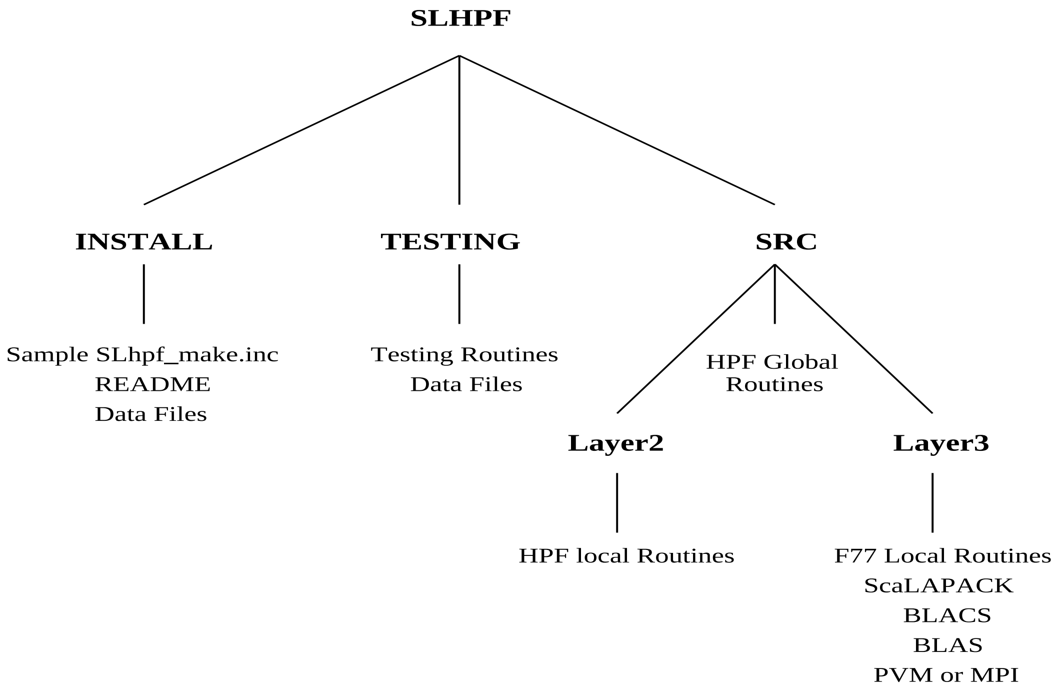

Figure 1: Organization of SLHPF

```
http://www.netlib.org/scalapack/prototype/index.html
```

Or, you can retrieve the file via anonymous ftp at `netlib`:

```
ftp ftp.netlib.org
login:  anonymous
password:  <your email address>
cd scalapack/prototype
binary
get slhpf.tar.gz
quit
```

The software on the `tar` file is organized into a number of essential directories as shown in Figure 1. Please note that this figure does not reflect every file contained in the `SLHPF` directory. Libraries are created in the SLHPF directory and executable files are created in TESTING directory. Input files for the testing programs are also found in the testing directory, so that testing may be performed in the directory SLHPF/TESTING. A top-level makefile in the SLHPF directory is provided to perform the entire installation procedure.

4 Overview of Library Contents

Most routines in SLHPF occur in four versions: REAL, DOUBLE PRECISION, COMPLEX, and COMPLEX*16. The first three versions (REAL, DOUBLE PRECISION, and

COMPLEX) are written to interface with standard Fortran 77 and are completely portable; the COMPLEX*16 version is provided for those Fortran 77 compilers which allow this data type. For convenience, we often refer to routines by their single precision names; the leading 'S' can be replaced by a 'D' for double precision, a 'C' for complex, or a 'Z' for complex*16. The module `HPF_LAPACK` provides an interface so that the top level driver routines do not have an S D C or Z prefix, but rather have the prefix `LA_`. There is a separate tester for each precision of the driver routines.

4.1 SLHPF Routines

There are three classes of SLHPF routines:

- **Global** HPF routines that are located in the SLHPF/SRC directory that include the actual interface routines and the HPF module for interfacing with HPF programs. These routines are all HPF global and will only call HPF global and HPF local routines. We will refer to these as Layer 1 routines.
- **Local** HPF routines which are called by the Global routines. These routines are used by the interface to call Layer 3 and ScaLAPACK's F77 routines. They also perform some local tasks. We will refer to these as Layer 2 routines.
- **Layer 3** routines are F77 routines called by the Layer 2 routines. The ScaLAPACK library, along with the BLACS, PVM, or MPI are considered to be at this level.

4.2 SLHPF Test Routines

This release contains test programs, located in SLHPF/TESTING, for each of the interface routines in each data type. These test programs test different types of distributions to ensure that the interface is installed correctly. The test program executables will have the form of x*p*routinename where *p* is the precision and routinename is the name of the driver routine such as gesv. For example `xsgesv` is the single precision tester for gesv. Data files for the testers are provided in the same directory. It is assumed that the ScaLAPACK test suites have already been run successfully. The SLHPF test suite should be run before using this interface in your own code.

5 Installing SLHPF on a Unix System

Installing and testing release version 1.1 of SLHPF involves the following steps:

1. Uncompress and tar the file.
2. Edit the file `SLHPF/SLhpf_make.inc`.
3. If using MPI with PGHPF, set the environment variable `HPF_MPI` and edit the file `SLHPF/SRC/misc.h`.
4. Type `make all`

5.1 Untar the File

If you received a tar file of SLHPF via the World Wide Web or anonymous ftp, enter the following command to untar the file:

`gunzip -c` *file* | `tar xvf -`

where *file* is the name of the gzipped tar file. This will create a top-level directory called `SLHPF`, which requires approximately 10 Mbytes of disk space. The total space requirements including the object files and executables is approximately 60 Mbytes for all four data types.

5.2 Edit the file `SLhpf_make.inc`

Before the libraries can be built, or the testing programs run, you must define all machine/compiler-specific parameters for the architecture and HPF compiler to which you are installing SLHPF. All machine-specific parameters are contained in the `SLhpf_make.inc` file. Sample `SLhpf_make.inc` files for different machines and/or compilers are in the `INSTALL` directory. Copy the one closest to your system to the `SLHPF` directory and rename it `SLhpf_make.inc`.

The first line of this `SLHPF_make.inc` file is:

TOPdir = $(HOME)/SLHPF

and may need to be modified to wherever you have put your SLHPF directory. Second, you will need to modify the `PLAT` definition to specify the architecture to which you are installing SLHPF. Next, you will need to modify `SLdir`, `Bdir`, `Mpdir`, `Mplib`, `BLAS`, `SYSlib`, to indicate where the ScaLAPACK, BLACS, Message Passing (MPI or PVM or other message passing library) and BLAS libraries are located. The definition of `setup` allows you to specify whether you are running PVM, MPI, or any other message passing layer. You may need to modify `SYSlib` for specific system libraries that also need to be included. For example, Solaris requires you to link in the -lsocket and -lnsl libraries when using MPI or PVM. The CONVERT_FROM and CONVERT_TO allows you to convert from/to when changing what extrinsic declaration is used for Layer 3 routines. This depends on your HPF compiler. For example Portland Group HPF (PGHPF) requires f77_local for Layer 3 while Digital Equipment requires hpf_local. Finally you will need to set your compiler and flags for each layer as well as the archiver flags. These can be set with `FL1`, `FL1FLAGS`, `FL2`, `FL2FLAGS`, `FL3`, `FL3FLAGS`, `L1LOADER`, `L1LOADFLAGS`, `ARCH`, `ARCHFLAGS`, and `RANLIB`. For more details on the SLhpf_make.inc file please see the `README` file in the `SLHPF/INSTALL` directory, and the `errata.SLHPF` file on netlib (URL address: `http://www.netlib.org/scalapack/prototype/errata.SLHPF`).

5.3 If you are using MPI under PGHPF

If you are using MPI under PGHPF you need to set the environment variable `HPF_MPI` to point to your MPI library, for example

```
setenv HPF_MPI /usr/local/MPI/mpich/lib/solaris/ch_p4/libmpi.a
```

You will also need to modify `SLHPF/SRC/misc.h` to change `EXITVAL=0` to `EXITVAL=1` if your compiler is using the same message passing layer as the BLACS you are running. Make sure that you have set `setup` to MPI in your `SLhpf_make.inc` file.

5.4 Compiling SLHPF

To compile the library and testing files:

```
cd SLHPF

make all
```

Or just compile the library by:

```
cd SLHPF

make lib
```

The removal of object files can be accomplished by the following:

```
cd SLHPF

make clean
```

For further installation details, please see the README file in the `SLHPF/INSTALL` directory.

6 Run the SLHPF Test Programs

6.1 A Simple Example Program

Included in the SLHPF/TESTING directory is the simple example program `xsimple`. This is the first program you should try to run. If this runs correctly, then you can proceed to running the entire SLHPF test suite and then use SLHPF in your own HPF 1.1 programs. Another feature of the simple example program code is that its source, `SLHPF/TESTING/xsgesv.f`, is very simple and easy to read. It will serve as a good guide of how to call an SLHPF routine by using the SLHPF module `HPF_LAPACK`. In this particular code we are calling a dgesv solver and comparing the results with the actual matrix that is obtained from matmul. The other testers are designed to test the interface to make sure that the implementation of SLHPF is correctly installed and are more difficult to read.

```
      program simplegesv
!
!  -- Layer 1 ScaLAPACK HPF wrapper routine, (version 1.1) --
!     September 1, 1998
!     Written by R. Clint Whaley, University of Tennessee, Knoxville
!
      use HPF_LAPACK
      integer, parameter :: N=500, NRHS=20, NB=64, NBRHS=64, P=1, Q=4
      integer, parameter :: DP=kind(0.0D0)
      integer :: IPIV(N)
      real(DP) :: A(N, N), X(N, NRHS), B(N, NRHS)
!HPF$ PROCESSORS PROC(P,Q)
!HPF$ DISTRIBUTE A(cyclic(NB), cyclic(NB)) ONTO PROC
!HPF$ DISTRIBUTE (cyclic(NB), cyclic(NBRHS)) ONTO PROC :: B, X

!
!     Randomly generate the coefficient matrix A and the solution
!     matrix X.  Set the right hand side matrix B such that B = A * X.
!
      call random_number(A)
      call random_number(X)
      B = matmul(A, X)
!
!     Solve the linear system; the computed solution overwrites B
!
      call la_gesv(A, B, IPIV)
!
!     As a simple test, print the largest difference (in absolute value)
!     between the computed solution (B) and the generated solution (X).
!
      print*,'MAX( ABS(X~ - X) ) = ',maxval( abs(B - X) )
!
!     Shutdown the ScaLAPACK system, I'm done
!
      call SLhpf_exit()

      stop
      end
```

To run `xsimple` using MPI run your program as though you were running any other MPI executable on your system. For example if you were using *mpich* you would use the following in the `SLHPF/TESTING` directory:

```
% mpirun xsimple -procs 4
```

To run `xsimple` using PVM you would run your program as though you were running any other HPF executable on your system. Note: you need to start up PVM before running your executable. For example with Portland Group HPF you would use the following in the `SLHPF/TESTING` directory:

```
% xsimple -pghpf -np 4
```

The output should look something like the following:

```
% xsimple -pghpf -np 4
MAX( ABS(X~ - X) ) =   2.5913715617775779E-012
FORTRAN STOP
```

If the difference is around 1.0E-8 or less then your program is working correctly and has passed the test within a double precision tolerance.

6.2 Testers

There are test programs for each of the interface routines in each data type in the TESTING directory. There is an input file for each of the testing programs. These testers are designed to check that the HPF data distribution is working correctly, and assume that ScaLAPACK, BLAS, and BLACS have already been tested and are working correctly. The tests have real, complex, double, and double complex routines unless otherwise indicated. The following is a list of the testers for the REAL version.

```
xsgels

xsgemm

xsgesv

xsimple (simple example program)

xsposv

xssyev (real and double precision only)

xstrsm
```

For the other precisions the leading 'xs' file names must be changed to 'xc', 'xd', or 'xz'. If you encountered failures in this phase of the testing process, please refer to Section 6.3. For information on the data files and how to modify them please see the `SLHPF/INSTALL/README` file for a detailed description.

6.3 Send the Results to Tennessee

Congratulations! You have now finished installing, and testing SLHPF. If you encountered failures in any phase of the testing, please consult the README file in the SLHPF/INSTALL directory and our `errata.SLHPF` file on netlib (URL address: `http:www.netlib.org/scalapack/prototype/errata.SLHPF`). This file contains machine-dependent installation clues which hopefully will alleviate your difficulties or at least let you know that other users have had similar difficulties on that machine. If there is not an entry for your machine or the suggestions do not fix your problem, please feel free to contact the authors at

`scalapack@cs.utk.edu`.

Tell us the type of machine on which the tests were run, the version of the operating system, the compiler and compiler options that were used, and details of the ScaLAPACK, BLACS, and BLAS libraries that you used. You should also include a copy of the output file in which the failure occurs. We encourage you to make the SLHPF library available to your users and provide us with feedback from their experiences. This release of SLHPF is not guaranteed to be compatible with any previous test release.

7 Required HPF features

The approach we are using requires the following HPF features:

1. `!HPF$ INHERIT`
   Used to ensure no unneeded matrix redistribution occurs across subroutine calls

2. `HPF_DISTRIBUTION` (from HPF_LIBRARY)
   Used to determine the distribution of the matrix's ultimate align target

3. `HPF_ALIGNMENT` (from HPF_LIBRARY)
   Used to determine how the distribution of the ultimate align target affects the actual matrix distribution

4. `!HPF$ PROCESSORS PROC(P,Q)`,
   where `P` and `Q` are dummy arguments to a routine
   When we have determined that redistribution must occur, this command is used to ensure that all matrices passed to the routine are distributed over the same process grid

5. `!HPF$ ALIGN X(:)  WITH A(:,*)`
   This command is used to ensure vectors without proper distributions are aligned correctly to their corresponding arrays that are already correctly distributed, or in some cases used as `!HPF$ ALIGN X(*) WITH A(*,*)` *to replicate a vector across all processors*

6. `!HPF$ DISTRIBUTE A(CYCLIC(MB) CYCLIC(NB)) ONTO PROC`
   `MB` and `NB` parameters to routine, `PROC` from above

When we have determined that a redistribution must occur, this allows us to express the kind of distributions ScaLAPACK will accept

7. HPF_LOCAL_LIBRARY
   Used for labeling the processes in a system independent way, allowing for a system independent process grid setup

8. A way to go from HPF to a language such as Fortran77. We will need the following extrinsics:

   - HPF_LOCAL
     Gives us access to HPF_LOCAL_LIBRARY
   - F77_LOCAL, F90_LOCAL, or HPF2.0's HPF_LOCAL
     Declares our Fortran77 library, so it can take assumed size arrays

8 Further details of the interface

Our wrapper library is divided into three distinct layers. Layer 1 is the global HPF layer, consisting solely of strict HPF code. It never makes calls to Layer 3; all such calls are routed through Layer 2. Layer 3 is the Fortran77 message passing layer, containing strict Fortran77 code (or C code made to be callable from Fortran77). This layer contains all the Fortran77 routines used, including the ScaLAPACK library, as well as some tool routines for the SLHPF wrapper library.

Layer 2 is the transition layer, existing in order to facilitate the transition from the global HPF layer to the local Fortran77 message passing layer. This layer is HPF_LOCAL code.

8.1 Layer 1

Layer 1 represents the user-callable wrapper functions, and their HPF tools. This layer is responsible for accepting the user's arguments, ensuring they are in a format which ScaLAPACK supports, and calling the appropriate Layer 2 wrapper to the ScaLAPACK routine. They should also be written in such a way as to minimize data movement.

To implement this, all wrapper routines have two paths to making a ScaLAPACK call. The first is a direct call to the appropriate Layer 2 routine, with all operands having the INHERIT attribute. This path requires no data movement, and if the compiler is sophisticated, should avoid copying the matrices as well.

The following things must be true for this optimal path to be followed:

- All matrix operands can be expressed as some legal form of CYCLIC(K) distribution (this includes all BLOCK distributions).
- All matrix operands are distributed across the same process grid (or, in some degenerate cases, some subset of the same process grid).
- The process grid over which the matrix operands are distributed over is one or two dimensional.

- Certain routine-dependent alignment restrictions between matrix operands are met.

If any of the above assertions are not true, we cannot use the most optimal path to calling ScaLAPACK, and must instead force a redistribution of the data to a form which ScaLAPACK can support. In this case, we call a Layer 2 wrapper to the ScaLAPACK routine, which contains explicit distribution instructions to guarantee all of the above assertions hold true. The `REDIST` flag in the `SLHPF/SRC/misc.h` file can be set to `TRUE` to issue a warning whenever this second option is used, allowing the user to monitor if he will be taking a performance loss on his current data distribution.

8.2 Layer 2

This layer is responsible for the transition from a global HPF code to a local message passing library. This obviously involves the use of the `EXTRINSIC` features of HPF. In our wrappers, this layer is `EXTRINSIC(HPF_LOCAL)` code. This allows us to use the HPF_LOCAL_LIBRARY routines necessary for process grid formation. This intermediate level is also responsible for translating HPF's global assumed *shape* arrays to Fortran77's local assumed *size* arrays. See section 11 for further discussion of this issue.

In general, this layer contains two routines for every ScaLAPACK routine. One accepts INHERITed matrix operands for maximal performance, and the other accepts matrices which have been explicitly distributed. This layer also contains wrappers around some miscellaneous ScaLAPACK routines which need to be called from Layer 1, such as those responsible for initializing the process grid.

8.3 Layer 3

As mentioned before, this layer consists primarily of the ScaLAPACK library. However, there are several tool routines as well. These routines perform such functions as setting up the ScaLAPACK process grid, etc. Layer 3 is written in strict Fortran77, so the most natural extrinsic is `F77_LOCAL` or `HPF_LOCAL` when supported as in the HPF 2.0 standard. Since Fortran77 is a proper subset of Fortran90, this layer may also be declared `EXTRINSIC(F90_LOCAL)` if `F77_LOCAL` or `HPF_LOCAL` do not exist. *NOTE: The declaration of Layer 3 routines may change depending on the compiler and the extrinsics supported. See the* `SLHPF/INSTALL/README`*'s section which covers converting Layer 3 to a new extrinsic for details of how to change the extrinsic to match your compiler.*

In the HPF 1.1 standard, HPF_LOCAL routines are constrained to accepting only assumed shape arrays as arguments. Since Fortran77 requires assumed size, we are unable to declare these routines HPF_LOCAL under this definition. HPF 2.0 indicates that HPF_LOCAL routines which are not called from global HPF directly (true for all of layer 3) can accept assumed size arrays. Therefore, on compilers supporting HPF 2.0's definition of HPF_LOCAL, we can declare layer 3 routines to be `EXTRINSIC(HPF_LOCAL)`. An example of this is the DEC f90 compiler, which does not support F90_LOCAL or F77_LOCAL, but does support HPF_LOCAL routines accepting assumed size arrays. On this platform, all layer 3 routines are declared as `EXTRINSIC(HPF_LOCAL)`.

9 Explanation of needed HPF features

This section explains in more detail how and why we used certain features of HPF.

9.1 `!HPF$ INHERIT`

In order for the SLHPF library to be general purpose, it should accept any kind of legal input. Further, if it is to be used, it must show better performance than the user can easily obtain by writing the code himself. In light of this, we use the `!HPF INHERIT` directive as often as possible to avoid unnecessary redistribution of data. Redistribuition has two major drawbacks: memory usage, and performance degradation.

If a matrix must be redistributed before an operation, then obviously a new matrix must be allocated to store the redistributed matrix. Since matrices usually represent most of a linear algebra program's memory requirements and users often move to parallel computing because the problem is too big to fit in serial memory, this cost can quickly become burdensome. This affect is magnified when we consider routines which take multiple matrices, such as matrix multiplication (which takes a total of 3 matrices). If all of these matrices are redistributed, we must have sufficient memory for 6 N^2 arrays. This then reduces the maximal size of the problem we can solve, which tends to keep the $O(N^2)$ communication term significant.

Redistribution also leads to performance degradation because the cost of communication is much greater than the cost of computation. This is true even on dedicated parallel machines; on clusters of workstations there may be orders of magnitude difference in computation and communication speeds. It is therefore obvious that there must be much more computation than there is data movement to justify a redistribution cost. If we are required to perform a redistribution when envoking library calls, we see that routines which have operation counts of the same order of magnitude as their data (e.g., the level 1 and 2 BLAS), will be prohibitively expensive. However, there is a large set of linear algebra routines which have $O(N^3)$ operations, while having only $O(N^2)$ data. With these routines, one may hope to be better able to tolerate the redistribution costs, since the computation time should dominate.

It is obviously true that we will need the size of our computation to be fairly large if we are going to have our $O(N^3)$ computation dominate our $O(N^2)$ communication time, bearing in mind the relative costs of these commodities. This is where the fact that redistribution consumes more memory also becomes a factor. Suffice it to say, that for many operations, the cost of the redistribution will be much greater than the cost of the operation itself.

Using INHERIT, we can ensure that no unnecessary redistribution occurs, both when the user passes the matrix to us, and when we pass it internally. In cases when the matrix operands are correctly distributed, the INHERIT command assures no data movement will occur. In the case when we must redistribute, INHERIT ensures that no data movement is required as we pass the matrices through our intermediate routines.

9.2 `HPF_DISTRIBUTION` and `HPF_ALIGNMENT`

`HPF_DISTRIBUTION` is used to find the rank and shape of the process arrangement over which a matrix is distributed. A process arrangement is acceptable if it is one or two dimensional. These arrangements correspond to a PxQ process grid, where P and Q are returned in the process shape.

`HPF_DISTRIBUTION` also returns the distribution information of the ultimate *align-target* of the matrix. In our terms we discover whether the distribution corresponds to a legal `CYCLIC(k)` mapping. Please note that `BLOCK` and even `COLLAPSED` dimensions may be expressed as `CYCLIC(k)` distributions. Also, this distribution information does not necessarily describe the distribution of the matrix we are concerned with: it describes the distribution of the ultimate *align-target* of the matrix. To determine how the information returned by `HPF_DISTRIBUTION` affects the matrix, we must call `HPF_ALIGNMENT`.

In the example routines, our use of `HPF_DISTRIBUTION` and `HPF_ALIGNMENT` is confined to the routine `SLhpf_dmatinf`.

9.3 `!HPF$ PROCESSORS PROC(P,Q)` and `!HPF$ DISTRIBUTE A(CYCLIC(MB) CYCLIC(NB)) ONTO PROC`

When we have determined that the matrix operands need to be redistributed, we must call an explicit interface which guarantees a particular distribution that ScaLAPACK can handle. We use these two statements for this purpose.

ScaLAPACK requires that all matrix operands be distributed over the same process grid. Our use of `PROCESSORS` and the `ONTO` clause of `DISTRIBUTE` guarantee this. To see an example of our usage of this feature, examine the interface section of `SLhpf_dgesv`.

9.4 `!HPF$ ALIGN WITH`

The `!HPF$ ALIGN WITH` is used in the wrappers in two different ways. First of all we use `!HPF$ ALIGN X(*) WITH A(*,*)` to replicate the vector X amongst all of the processors used by A. The second way we use this is to align a particular vector to a matrix that already has the correct distribution; this is only necessary if the vector is not already correctly distributed. `!HPF$ ALIGN WITH` allows us to correctly correlate the distributions of vectors with their corresponding matrices.

9.5 Labeling the processes for process grid setup

As mentioned before, ScaLAPACK has its own message passing layer, the BLACS. The BLACS take as input the process IDs that define a particular process grid upon which the ScaLAPACK computation is to take place.

Therefore, in order to get ScaLAPACK started, we will need to find the process grid over which the matrix has been distributed. To do this, we need to find in what process's memory particular blocks of the matrix reside. Routines from HPF_LOCAL_LIBRARY are used to establish this mapping.

10 Implementation details

This section provides a quick overview of the main routines used in the wrappers. Figure 2 shows a simplified version of the internal calls that occur when a user calls `LA_GESV` with double precision arguments. This schematic flow chart establishes the hierarchy of the individual routines and complements their descriptions in the following subsections.

10.1 Layer 1

SLhpf_dgesv HPF interface to `p□gesv`.

SLhpf_ddescset3 Determines if the matrices can be represented by a simple cyclic(k) format. If they can, fills in descriptor (including formation of context), and returns `INFO = 0`. Otherwise, the context is not formed, and `INFO` is returned as nonzero.

SLhpf_dmatinf Fills in the descriptor entries `M_`, `N_`, `MB_`, `NB_`.

SLhpf_dgridinf Fills in a gridmap describing the process grid over which the matrix is distributed.

10.2 Layer 2

SLhpf_dget_procmap Determines the gridmap of the process grid over which the matrix is distributed. The HPF_LOCAL_LIBRARY routine ABSTRACT_TO_PHYSICAL gives the gridmapping in terms of HPF IDs. The HPF_LOCAL_LIBRARY routine GLOBAL_TO_LOCAL gives the process coordinates having the first block of the matrix. These IDs are translated into the process IDs used by the BLACS through a mapping established by `SLhpf_get_hpf2sys_map`.

SLhpf_compare_grids Establishes whether or not two grids are conformant. Two grids are conformant if they are equivalent or one is an equivalent subset of the other. Two grids are equivalent if they have the same number of rows and columns (i.e. they are both $r \times c$ grids, and entry (i, j) in grid A is the same process as entry $((RSRC+i) \bmod r), ((CSRC+j) \bmod c)$, where $RSRC$ and $CSRC$ are constants whose value is $0 \leq RSRC < r$ and $0 \leq CSRC < c$.

If a grid is an equivalent subset of another grid, its dimensions are less, its matrix is not overdecomposed, and the above entry relationship holds.

SLhpf_get_context Once a gridmap shared by all operand matrices has been filled in, this routine calls the appropriate BLACS routines to form a context corresponding to the required grid. This routine caches the last context formed. If the new grid is conformant with that of the cached context, the old context

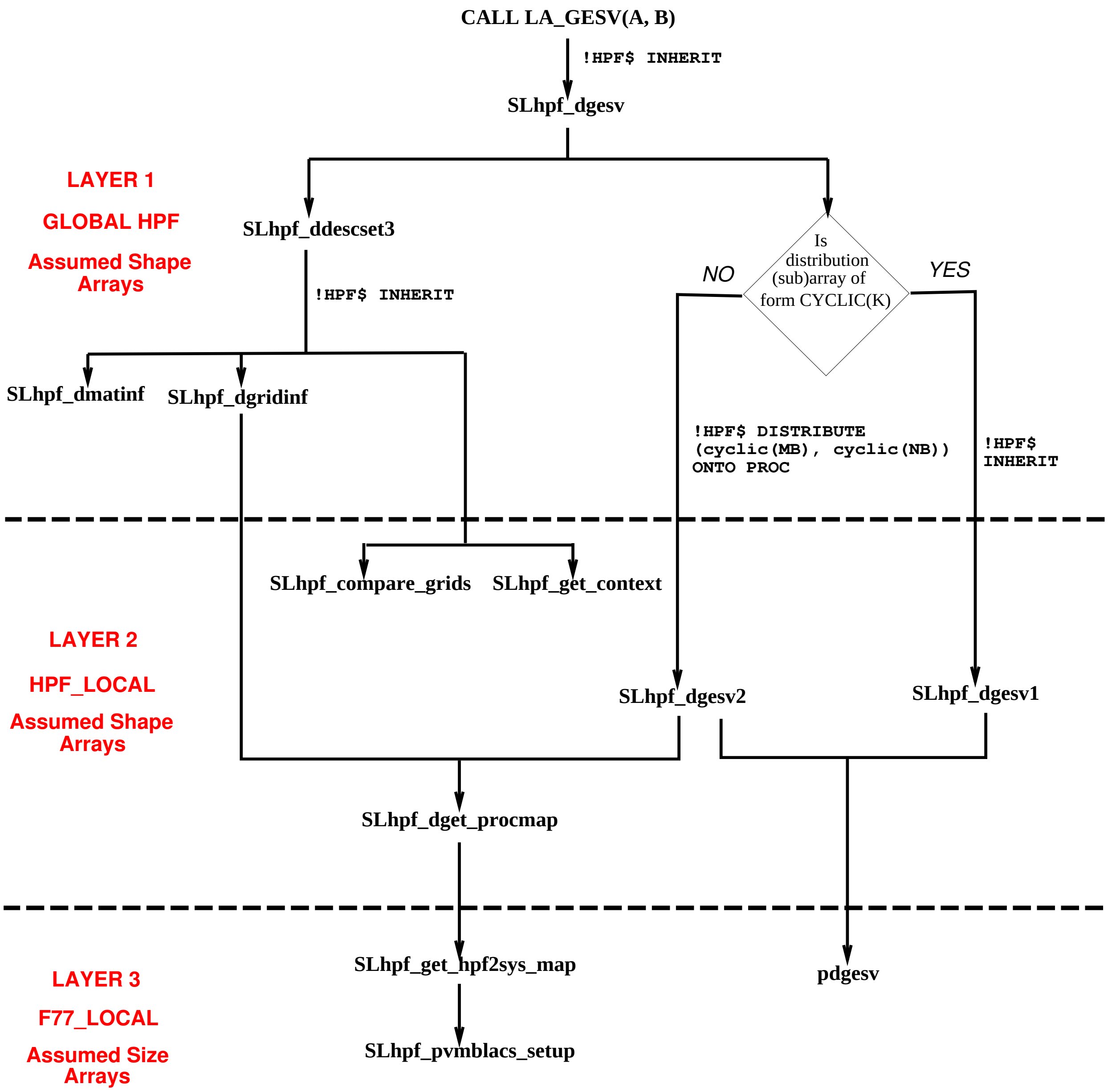

Figure 2: Call hierarchy

is reused. Otherwise, the old context is freed, and the new one is cached for the next call.

SLhpf_dgesv1 Intermediary interface to `pdgesv`. This wrapper INHERITs all matrices, so no redistribution will be done.

SLhpf_dgesv2 Intermediary interface to `pdgesv`. This wrapper is called when a redistribution is required. After redistribution has taken place, this routine calls `SLhpf_dget_procmap` to find the new grid and calls the BLACS directly to form the context.

10.3 Layer 3

SLhpf_get_hpf2sys_map At the cost of communication of order $2 \log p$ (where p is the number of processes in the HPF application), this routine establishes a mapping between HPF process IDs and those used by the BLACS. Note that this mapping does not change, so this routine is called only once, and thus the communication cost can be amortized over all calls to the wrapper library. If the PVMBLACS are being used, it calls `SLhpf_pvmblacs_setup` in order to dynamically form a PVM machine from the HPF-launched processes.

SLhpf_pvmblacs_setup Called only when using PVMBLACS. Dynamically forms a PVM machine from HPF-launched processes. Called only once for each execution.

SLhpf_mpiblacs_setup Stub routine.

pdgesv ScaLAPACK routine for solving a general system of linear equations.

11 Extrinsics

As mentioned previously, our codes are written in Fortran77 and C written to be callable from Fortran77. We therefore need a way to go from the HPF code, which contains global descriptions of the matrix, to a local Fortran77 message passing view of the matrix. Obviously, since this represents a change in language, we must make use of HPF's `EXTRINSIC` declaration.

At first glance, the obvious extrinsic is `F77_LOCAL`. After all, this is what our routines are. However this overlooks an important area of interfacing HPF routines and local Fortran77.

```
SUBROUTINE F90_TO_F77(A)
REAL, INTENT(INOUT) :: A(:,:)

INTEGER :: M, N, LDA

INTERFACE
   SUBROUTINE F77ROUT(M, N, A, LDA)
   INTEGER, INTENT(IN) :: M, N, LDA
   REAL, INTENT(INOUT) :: A(LDA,*)
   END SUBROUTINE F77ROUT
END INTERFACE

M = SIZE(A, 1)
N = SIZE(A, 2)
LDA = M

CALL F77ROUT(M, N, A, LDA)

RETURN
END
```

Figure 3: A F90 routine passing a 2D array to a F77 routine

The problem involves the translation of a HPF/F90 assumed *shape* array to a Fortran77 assumed *size* array.

Fortran77 arrays are local arrays of assumed size. An assumed size array is basically just a memory address, where all dimensions of the matrix save the last are specified so that the compiler can do the proper indexing. This, then, is the problem that faces us when going from HPF's global view of the matrix to Fortran77: finding out the extent of the *local* 2D array's first dimension so that we may do our index arithmetic. We refer to this extent as the local leading dimension (LLD for short) of the matrix.

This problem may be overcome in F90 by specifying an explicit interface. This approach is illustrated in figure 3. Obviously, if the compiler has not already stored the indicated (sub)array in an array whose first dimension is equal to the `SIZE` of the array, it will need to allocate space and copy the array.

Note that in distributed memory terms, the leading dimension (`LDA` in the above example) is an inherently *local* quantity: it indicates the memory stride between elements in a row. Thus the F90 approach cannot be directly utilized in HPF, since the interface described by HPF is *global*.

This oversight means that only 1D arrays may be standardly passed from HPF to external languages such as C or Fortran77 (this is obviously true because it is impossible to determine how the compiler has locally laid out the matrix, and thus it will be impossible to do indexing on such arrays). This means that codes wishing to pass arrays with greater than 1 dimension will have to be compiler specific (actually, compiler version specific).

Compilers supporting EXTRINSIC(F90_LOCAL) should not have this problem. There, the user may pass the HPF matrix to a F90_LOCAL routine which takes the matrix as assumed shape. The F90_LOCAL routine may then use the code in figure 3 to pass the local array to the Fortran77 code.

```
EXTRINSIC(HPF_LOCAL) &
SUBROUTINE HPFLOC_TO_F77(A)
REAL, INTENT(INOUT) :: A(:,:)

INTEGER :: M, N, LDA

INTERFACE
   EXTRINSIC(F77_LOCAL) &
   SUBROUTINE F77ROUT(M, N, A, LDA)
   INTEGER, INTENT(IN) :: M, N, LDA
   REAL, INTENT(INOUT) :: A(LDA,*)
   END SUBROUTINE F77ROUT
END INTERFACE

M = SIZE(A, 1)
N = SIZE(A, 2)
LDA = M

CALL F77ROUT(M, N, A, LDA)

RETURN
END
```

Figure 4: A HPF LOCAL routine passing a 2D array to a F77 routine

A similar solution may be used if a language supports both HPF_LOCAL and F77_LOCAL. HPF_LOCAL routines are mandated to accept only assumed shape arrays, so this extrinsic alone cannot solve the problem. However, in conjunction with F77_LOCAL, we can adopt the solution much like the one above. This solution is shown in figure 4. Note: if the compiler supports HPF_LOCAL as in the HPF 2.0 standard, then one can use HPF_LOCAL in place of F77_LOCAL.

Acknowledgments

We acknowledge with gratitude the support which we have received from the following organizations, and the help of individual members of their staff: Portland Group; Digital Equipment Corporation.

Appendix A

Calling sequences

Users wishing to call ScaLAPACK routines need to USE the module HPF_LAPACK. The file SLHPF/TESTING/simple_gesv.f shows a simple call to LA_GESV. The calling sequences for the supplied routines are (default values for optional parameters are enclosed in []):

```
SUBROUTINE LA_GESV(A, B, IPIV, INFO)
     <TYPE>, INTENT(INOUT), DIMENSION(:,:) :: A, B
     INTEGER, OPTIONAL, INTENT(OUT) :: IPIV(:), INFO

  SUBROUTINE LA_POSV(A, B, UPLO, INFO)
     <TYPE>, INTENT(INOUT), DIMENSION(:,:) :: A, B
     CHARACTER(LEN=1), OPTIONAL, INTENT(IN) :: UPLO[='Upper']
     INTEGER, OPTIONAL, INTENT(OUT) :: IPIV(:)

  SUBROUTINE LA_GELS(A, B, TRANS, INFO)
     <TYPE>, INTENT(INOUT), DIMENSION(:,:) :: A, B
     CHARACTER(LEN=1), OPTIONAL, INTENT(IN) :: TRANS[='NoTranspose']
     INTEGER, OPTIONAL, INTENT(OUT) :: IPIV(:)

  SUBROUTINE LA_SYEV(A, W, Z, UPLO, INFO)
     <TYPE>, INTENT(INOUT), DIMENSION(:,:) :: A
     <TYPE>, INTENT(OUT), DIMENSION(:) :: W
     <TYPE>, OPTIONAL, INTENT(OUT), DIMENSION(:,:) :: Z
     CHARACTER(LEN=1), OPTIONAL, INTENT(IN) :: UPLO[='Upper']
     INTEGER, OPTIONAL, INTENT(OUT) :: INFO


  SUBROUTINE LA_GEMM(A, B, C, TRANSA, TRANSB, ALPHA, BETA)
     <TYPE>, INTENT(IN), DIMENSION(:,:)     :: A, B
     <TYPE>, INTENT(INOUT), DIMENSION(:,:)  :: C
     CHARACTER(LEN=1), OPTIONAL, INTENT(IN) :: TRANSA['NoTranspose'],
                                               TRANSB[='NoTranspose']
     <TYPE>, OPTIONAL, INTENT(IN) :: ALPHA[=1.0], BETA[=0.0]
```

```
SUBROUTINE LA_TRSM(A, B, SIDE, UPLO, TRANSA, DIAG, ALPHA)
   <TYPE>, INTENT(IN), DIMENSION(:,:)     :: A
   <TYPE>, INTENT(INOUT), DIMENSION(:,:)  :: B
   CHARACTER(LEN=1), OPTIONAL, INTENT(IN) :: SIDE[='Left'],
                                             UPLO[='Upper'],
                                             TRANSA[=NoTranspose'],
                                             DIAG[='NonUnit']
   <TYPE>, OPTIONAL, INTENT(IN) :: ALPHA[=1.0]
```

For more details, see the module file, `SLHPF/SRC/HPF_LAPACK_mod.f`.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, Second Edition, SIAM, Philadelphia, PA, 1995.

[2] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J.Demmel, I. Dhillon, J.Dongarra, S. Hammarling, G.Henry, A.Petitet, K. Stanley, D. Walker, R.C. Whaley *ScaLAPACK Users' Guide*, SIAM, Philadelphia, PA, 1997.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine. A Users' Guide and Turtorial for Networked Parallel Computing*, MIT Press, Cambridge, MA 1994.

[4] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge, MA 1996.

[5] C. Koebel, D. Loveman, R. Schreiber, G.Steele, M. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA 1994.

[6] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft.*, 16, 1:1-17, March 1990.

[7] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs," *ACM Trans. Math. Soft.*, 16, 1:18-28, March 1990.

[8] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft.*, 14, 1:1-17, March 1988.

[9] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms: Model Implementation and Test Programs," *ACM Trans. Math. Soft.*, 14, 1:18-32, March 1988.

[10] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Soft.*, 5, 3:308-323, September 1979.