HPF and Numerical Libraries*

Harald J. Ehold[1], Wilfried N. Gansterer[2], Dieter F. Kvasnicka[3], and Christoph W. Ueberhuber[2]

[1] VCPC, European Centre for Parallel Computing at Vienna
ehold@vcpc.univie.ac.at
[2] Institute for Applied and Numerical Mathematics, Vienna University of Technology
ganst@aurora.tuwien.ac.at, christof@uranus.tuwien.ac.at
[3] Institute for Physical and Theoretical Chemistry, Vienna University of Technology
dieter@titania.tuwien.ac.at

Abstract. Portable and efficient ways for calling numerical high performance software libraries from HPF programs are investigated. The methods suggested utilize HPF's EXTRINSIC mechanism and are independent of implementation details of HPF compilers. Two prototypical examples are used to illustrate these techniques. Highly optimized Blas routines are utilized for local computations: (i) in parallel multiplication of matrices, and (ii) in parallel Cholesky factorization. Both implementations turn out to be very efficient and show significant improvements over standard HPF implementations.

1 Introduction

High Performance Fortran (HPF [11]) is one of the most interesting approaches for high-level parallel programming. In particular, it provides very convenient ways for specifying data distributions and for expressing data parallelism. Development of parallel code using HPF is much easier and requires less effort than message passing programming, for example, using MPI.

However, in numerical applications the performance achieved with HPF programs is often disappointing compared to message passing code. This is partly due to the immaturity of HPF compilers, which can be explained by the difficulties to implement the advanced features of HPF efficiently. There is, however, another important aspect, which is often ignored. In order to achieve high performance it is crucial to integrate highly optimized library routines into HPF code.

Much effort has been spent on developing highly efficient implementations of the Basic Linear Algebra Subroutines (Blas [12,7,8]) and on numerical libraries for dense linear algebra which use the Blas as building blocks. Important examples are Lapack [1], which is the standard sequential library for dense or banded linear algebra methods, and parallel libraries such as ScaLapack [4] or PLapack [10]. Recently, code generation tools have been developed (see Bilmes

* This work was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund FWF.

P. Zinterhof, M. Vajteršic, A. Uhl (Eds.): ACPC'99, LNCS 1557, pp. 140–152, 1999.
© Springer-Verlag Berlin Heidelberg 1999

et al. [2], Whaley, Dongarra [15]) which automatically find the best choice of hardware dependent parameters for an efficient implementation of the BLAS.

The integration of such software into HPF is crucial for several reasons:

- In many scientific applications a lot of programming effort can be saved by using existing library routines as building blocks instead of (re-)coding them in HPF directly.
- A considerable amount of expertise has been incorporated into high quality software packages like the ones mentioned above. It is hardly possible to achieve comparable floating-point performance when coding in HPF directly. Even if a problem is well suited for an HPF implementation, the success of such an attempt heavily relies on the maturity of HPF compilers, which cannot be taken for granted at present (see Ehold et al. [9]).
- When standard numerical operations are coded in HPF the resulting code often suffers from poor local performance. Thus, one of the key issues is to optimize local performance in order to improve overall parallel performance.
- Highly optimized BLAS implementations are available for most target systems. Therefore, the use of BLAS routines for local computations ensures *performance portability.*
- Usually the main motivation for parallelization is performance improvement. It is essential to optimize the *sequential* code first in order to be able to measure the benefits of parallelization in a clean way. The sequential optimization typically involves restructuring the code, for example, by increasing the fraction of Level 3 (matrix-matrix) operations and by using appropriate BLAS 3 routines (or other high performance library routines) wherever possible. Thus, when the parallelization is done by using HPF, the necessity of combining the BLAS (and other numerical packages and libraries) with HPF arises quite naturally.

For all of these reasons the goal of this paper is to investigate ways to utilize high performance numerical libraries in an HPF context.

The basic facility provided by HPF for integrating procedures from other programming languages or models is the EXTRINSIC mechanism (HPF Forum [11]). This paper describes various methods, at different levels of abstraction, for calling existing library routines from HPF using this mechanism.

The techniques described are *portable* in the sense that they only rely on features from the HPF standard [11] and, additionally, on three of the HPF 2.0 Approved Extensions. In particular, the required HPF features are:

- an advanced form of the ALIGN directive, namely `ALIGN A(j,*) WITH B(j,*)` (replication of `A` along one dimension of `B`);
- the INHERIT directive;
- EXTRINSIC(HPF_LOCAL) subroutines;
- EXTRINSIC(F77_LOCAL) and EXTRINSIC(F77_SERIAL) subroutines.

Unfortunately, some HPF compilers do not yet support all of these features.

The main ideas and basic concepts presented in this paper are applicable to many numerical algorithms in dense linear algebra, but their implementation will

differ in various technical details. For the purpose of illustration two operations, which arise very frequently at the core of numerous scientific applications, serve as prototypes: Matrix-matrix multiplication and, representing more complicated numerical algorithms, Cholesky factorization.

In Section 2 ways for integrating sequential library routines into HPF code are suggested for these two operations. Section 3 gives an overview of techniques that have been used to interface HPF to the ScaLapack parallel library. In both cases experiments have been performed which demonstrate the considerable performance gains arising from the use of the techniques suggested.

2 Calling Sequential Routines from HPF

The topic of this section is the integration of extrinsic library routines into an HPF program for *local* computations only. All of the communication in the multiprocessor environment is organized by the HPF compiler.

A matrix-matrix multiplication routine (Section 2.1) and a Cholesky factorization routine (Section 2.2) were implemented. Both utilize the Blas for local computations. Experiments were performed with the HPF compiler from PGI, *pghpf*, version 2.4, on a Meiko CS-2 and an IBM SP2. The CS-2 experiments used a standard Fortran implementation of the Blas (Libblas), compiled with the *pgf77* Fortran 77 compiler from PGI, and Blas routines generated by the Atlas package (Whaley, Dongarra [15]). On the SP2 the vendor optimized Blas routines were utilized.

2.1 Multiplication of Matrices

As a first example, the computation of the product $C \in \mathbb{R}^{m \times n}$ of two matrices $A \in \mathbb{R}^{m \times l}$, $B \in \mathbb{R}^{l \times n}$ is considered where all matrices can have arbitrary distributions. An HPF routine called `par_dgemm` (*parallel* `BLAS/dgemm`) was developed, which performs the matrix-matrix multiplication. Internally, this operation is split up into local operations on subblocks, each of which is performed by calling the general matrix multiplication routine `BLAS/dgemm`.

In the general case, the multiplication of two distributed matrices involves *non-local* computations. Some of the data have to be replicated over several processors in order to localize all of the computations involved.

Loop Orders. The central operation in matrix-matrix multiplication is

$$C(i,j) = C(i,j) + A(i,k)B(k,j),$$

where $i = 1, 2, \ldots, m$, $j = 1, 2, \ldots, n$, and $k = 1, 2, \ldots, l$. Permuting the order of these three loops yields different algorithmic variants with different characteristics with respect to parallelization (and consequently, different requirements for expressing parallelism). The basic differences relevant to parallelization are as follows, where the variants are labeled by the index of the outermost loop.

k$*$ variant: The two inner loops perform the outer product of a column of A and a row of B. Since the entire matrix C is updated in place at every step of the outer loop, this variant exhibits no communication requirements for elements of C. Computation of a local block of C requires the corresponding parts of the columns of A and of the rows of B. In order to localize all of the computations, the columns of A have to be replicated in the direction of the rows of C, and the rows of B have to be replicated in the direction of the columns of C.

i$*$ variant: The two inner loops compute a row of C by multiplying the corresponding row of A with the columns of B. In this case it is usually most efficient to parallelize over both dimensions of the matrix B. This, however, requires distributed reduction operations for the elements of C.

j$*$ variant: The two inner loops compute a column of C by multiplying rows of A with the corresponding column of B. In this case it is usually most efficient to parallelize over both dimensions of the matrix A. This again requires distributed reduction operations for the elements of C.

The $k*$ variant requires the smallest amount of local storage without extra communication for the elements of C. This variant is well suited to a two-dimensional distribution of array C, whereas for the other two variants either a one-dimensional data distribution is to be preferred (columnwise for the $i*$ variant, rowwise for the $j*$ variant) or extra communication needs to be performed for the elements of C.

For load balancing reasons two-dimensional data distributions are normally used in linear algebra algorithms. In such a setup, the $k*$ variant exhibits the best data locality, minimizes the amount of data to be replicated, and leads to the most efficient parallelization. Consequently, it was chosen for the implementation.

Implementation. For performance reasons, a *blocked* version of matrix-matrix multiplication (Ueberhuber [14]) was implemented. Two levels of wrapper routines are involved. The programmer calls the HPF routine `par_dgemm`, which takes the matrices A, B as input and returns the product matrix C.

In this routine the HPF_LOCAL routine `local_dgemm` is called inside a loop. In addition to the blocks involved `local_dgemm` also takes their size (the block size of the algorithm) as an argument and performs the local outer products of a block column of A and a block row of B by calling the routine `BLAS/dgemm`.

In `par_dgemm`, work arrays for a block column of A and for a block row of B are aligned properly with C, and then the corresponding subarrays of A and B are copied there. This copying operation adjusts the distribution of the currently required parts of A and B to that of C. This copying is the only place where inter-processor communication occurs. It has the advantage of making the procedure fully independent of the prior distributions of A and B. All of the necessary communication, which is entirely contained in the copying operation, is restricted to the outermost loop.

Experiments. Fig. 1 illustrates the floating-point performance of `par_dgemm` and the vendor supplied intrinsic (parallel) function MATMUL for multiplying two $n \times n$ matrices on 16 processors of a Meiko CS-2. A and B were distributed cyclically in both dimensions, and C was distributed block-cyclically with block size 20 in both dimensions.

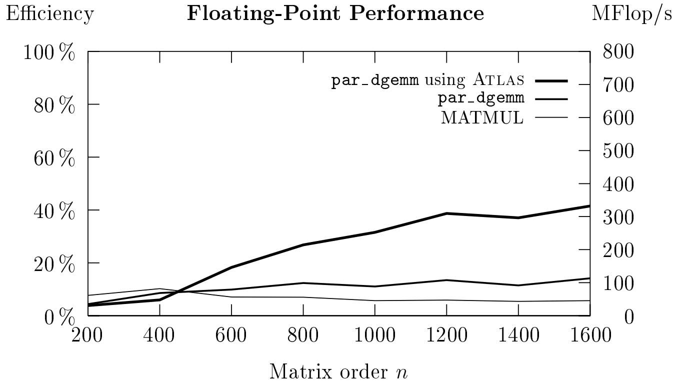

Fig. 1. Matrix multiplication on 16 processors of a Meiko CS-2. “Efficiency” denotes the percentage of the peak-performance achieved.

A considerable performance improvement of `par_dgemm` over MATMUL is evident for matrix orders $n > 500$ (roughly by a factor of 2 using the LIBBLAS, respectively 5 using the ATLAS-BLAS). This holds for other distribution scenarios as well.

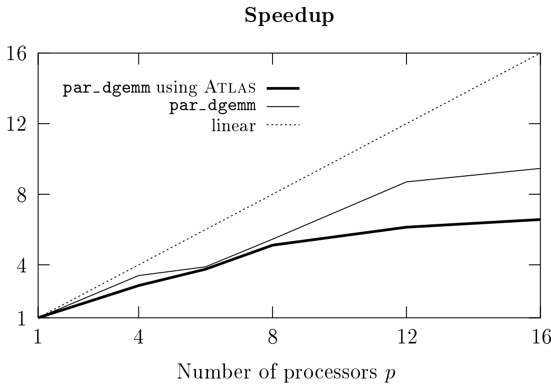

Fig. 2. Matrix multiplication on p processors for $n = 1000$ on a Meiko CS-2.

Moreover, for large matrix orders n the efficiency of par_dgemm improves, whereas the efficiency of the MATMUL routine decreases. The ATLAS generated BLAS/dgemm shows significant improvements in efficiency, especially for large matrices. It is important to keep in mind that 16 processors were used in the test run and that larger local blocks result in better utilization of the BLAS.

In order to evaluate parallel speedup (see Fig. 2), the runtime of the routine par_dgemm respectively par_dgemm using ATLAS on one processor was used as a reference value. The parallel speedup for up to 8 processors is quite the same for both par_dgemm routines. On more processors the ATLAS version does not scale so good anymore because the time for computation per processor is already so small that the communication becomes dominant.

Fig. 3 gives the efficiency on 1 to 16 processors of par_dgemm and MATMUL running on an IBM SP2. On the SP2 par_dgemm is based on the highly optimized ESSL/dgemm routine. The performance of par_dgemm scales reasonably well as long as the workload can be balanced well. MATMUL shows a very poor performance. MATMUL running on 16 processors of an IBM SP2 achieves a lower floating-point performance than ESSL/dgemm running on one processor. In this experiment it was not possible to test MATMUL running on one processor due to memory limitations.

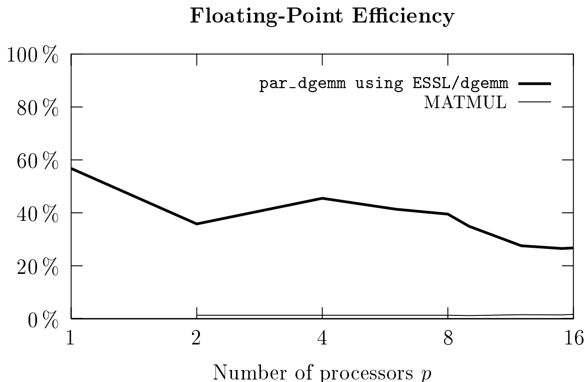

Fig. 3. Matrix multiplication on p processors for $n = 2000$ on an IBM SP2.

2.2 Cholesky Factorization

Even if the HPF compiler does a good job with organizing data distribution and communication between processors, the performance achieved can be disappointingly low due to bad node performance. Experiments have been performed with several HPF implementations of the same algorithm for computing the Cholesky factorization $A = LL^\top$ of a real symmetric positive definite matrix.

As shown in Table 1, using the sequential BLAS-based Cholesky factorization `LAPACK/dpotrf` is up to 12 times faster than the best scaling HPF code (Version 1) on one processor and still 1.3 times faster than this code on 16 processors. The HPF code which yields best sequential performance (Version 2) does not show any parallel speedup.

Table 1. Cholesky factorization of a matrix of order 1000, using the vendor optimized routine `LAPACK/dpotrf` running on one processor only, and two HPF versions on a Meiko CS-2.

p	1	2	4	8	16
`LAPACK/dpotrf`	**9 s**				
HPF Version 1	113 s	57 s	29 s	18 s	12 s
HPF Version 2	36 s	43 s	45 s	45 s	46 s

Utilizing BLAS Routines. One way of improving the nodal performance in HPF is to use the BLAS for doing the local computation on each node. The blocked algorithm for computing the factor L of the Cholesky factorization exhibits better locality of reference and is therefore to be preferred over the unblocked version on modern computer architectures (Anderson et al. [1], Ueberhuber [14]). A very important operation to be performed is the multiplication of certain submatrices of A.

The principle step of the blocked version of the Cholesky factorization is the following (see also Fig. 4):

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^\top & L_{21}^\top \\ 0 & L_{22}^\top \end{pmatrix} = \begin{pmatrix} L_{11}L_{11}^\top & L_{11}L_{21}^\top \\ L_{21}L_{11}^\top & L_{21}L_{21}^\top + L_{22}L_{22}^\top \end{pmatrix}.$$

This step requires the following three tasks.

1. Factorize $A_{11} = L_{11}L_{11}^\top$ using `LAPACK/dpotrf`.
2. Solve the linear system $L_{21}L_{11}^\top = A_{21}$ for L_{21} using `BLAS/dtrsm`.
3. Multiply $L_{21}L_{21}^\top$ using `BLAS/dgemm` and continue recursively with $A_{22} - L_{21}L_{21}^\top = L_{22}L_{22}^\top$.

In the HPF version the first task is done sequentially on one processor by using the HPF extrinsic kind F77_SERIAL, but the subsequent steps (`BLAS/dtrsm` and `BLAS/dgemm`) are parallelized. For simplicity it is assumed that the block size b divides the dimension of the matrix n evenly, where n is the size of the leading dimension of matrix A. The distribution chosen for the symmetric matrix A is (CYCLIC(b),$*$). The one-dimensional distribution was chosen to avoid communication within rows.

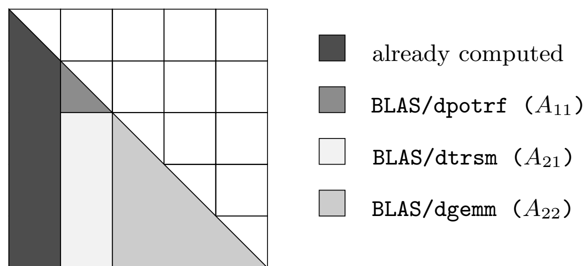

Fig. 4. Update areas of blocked Cholesky factorization.

Using HPF_LOCAL. The extrinsic kind HPF_LOCAL refers to procedures implemented in the HPF language, but in the *local* programming model. Its code is executed locally per processor and this gives the possibility to do local computations by calling the sequential BLAS routines.

In the parallel version, `BLAS/dtrsm` is called on each processor that owns local data of the block A_{21} (see Fig. 5). Hence in the example shown in Fig. 5 (two processors $P1$ and $P2$) there are two calls to `BLAS/dtrsm`. In this example processor $P1$ owns blocks B_1 and B_3 and needs block A_{11} and processor $P2$ owns block B_2 and also owns already block A_{11}. Since block A_{11} is needed by all participating processors it is broadcast to all of them. The block size on distinct processors can differ. For that a mechanism for determining the correct size of the local array is needed. Inside the HPF_LOCAL routine the intrinsic function SIZE gives the desired information that can be used by the BLAS routine.

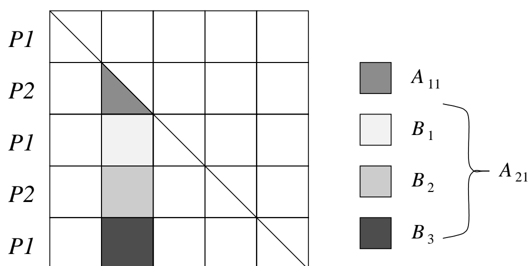

Fig. 5. Local blocks in parallel blocked Cholesky factorization.

The HPF_LOCAL version calls `BLAS/dtrsm` as an HPF_LOCAL routine by passing the appropriate array section to the subroutine. This section of the array needs to be passed to the HPF_LOCAL routine without changing the distribution of the array at the procedure boundary. In this paper two different methods were investigated to achieve this, using INHERIT or TEMPLATES, which are

both portable between all HPF compilers that support the core HPF 2.0 language.

1. The natural way to do this is using the INHERIT directive in the interface and the declaration of the HPF_LOCAL routine. INHERIT tells the compiler that an array and hence also subsections of this array have the same distribution in the subroutine as in the calling routine.
2. If an HPF compiler does not yet support INHERIT then TEMPLATES can be used to ensure that the distribution of array sections is not changed at a procedure boundary.

The drawback of this second version is that two additional arguments occur which are only needed for specifying the proper size and distribution of the template in the interface. One parameter gives the size of the first dimension of the array in the calling routine. This value is used to declare a TEMPLATE in the interface of the callee that has the same size and distribution as the starting array. A second parameter gives the starting point of the array section passed to the subroutine. With this information the dummy argument can be aligned to the corresponding part of the starting array via the use of the TEMPLATE.

Using F77_LOCAL. The extrinsic kind F77_LOCAL refers to procedures that are implemented in Fortran 77 and in the local programming model. This suggests that F77_LOCAL is the natural choice for combining HPF with the Blas, which use Fortran 77 calling conventions. However, since the Fortran 90 intrinsic function SIZE is typically not supported within a F77_LOCAL subroutine, different techniques have to be used for determining the size of local arrays. Another problem with this EXTRINSIC mechanism is that it is not clearly defined in the HPF standard (HPF Forum [11]), so there can be variations in the implementation details between different HPF compilers. For the HPF compiler used on the Meiko CS-2, INHERIT is not fully supported with F77_LOCAL, so no performance results can be given for this version.

Experiments. Runtimes on the Meiko CS-2 were the same for the HPF_LOCAL version using TEMPLATES as for the INHERIT version. For that reason Fig. 6 shows just one "HPF+Blas" curve.

The sequential routine `LAPACK/dpotrf` was benchmarked on one processor. The efficiency values for $p \geq 2$ processors were determined by simply dividing the single-processor value by p.

On one processor the superiority of the HPF+Blas versions over the pure HPF code can be seen. `ATLAS-LAPACK/dpotrf` yields very high efficiency on one processor. For an increasing number of processors a decreasing efficiency is to be observed. This phenomenon is due to the relatively small problem size ($n = 2000$) giving bad workload balancing for a one-dimensional processor distribution.

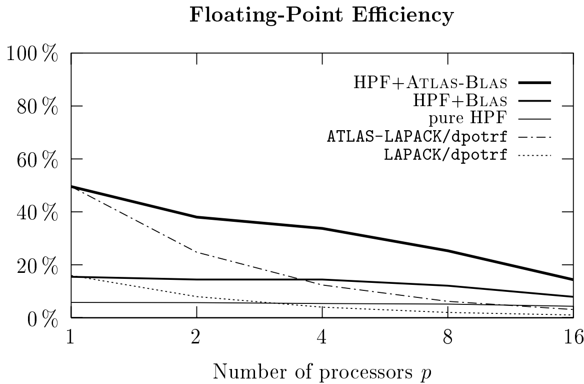

Fig. 6. Efficiency of the Cholesky factorization of a matrix of order $n = 2000$ on a Meiko CS-2.

3 Calling Parallel Libraries from HPF

As an alternative to the approach described in the previous section, extrinsic *parallel* library routines such as ScaLapack [4] and PLapack [10] can be invoked from HPF for distributed computations. In this case, the extrinsic library procedure itself performs communication as well as computation, and HPF is used as a framework for conveniently distributing data and for organizing high-level parallelism. Calling distributed library routines from HPF mainly involves the issue of passing distribution information from HPF to the library routines.

This section gives a brief survey of techniques that have been used to interface HPF to the ScaLapack parallel library.

3.1 Interfacing ScaLAPACK via EXTRINSIC Functions

Blackford et al. [5] have developed an interface called SLHPF from HPF to the ScaLapack package [4]. This interface uses three layers of wrapper routines: (i) global HPF routines, which call (ii) `HPF_LOCAL` routines, which in turn call (iii) Fortran 77 routines, which take local assumed-size arrays as arguments. The first version of the SLHPF interface contains HPF wrapper routines for LU and Cholesky factorization, and for the level 3 Blas routines `BLAS/dgemm` and `BLAS/dtrsm`, among others.

The wrapper routines do not have a significant influence on the execution time (see Blackford et al. [5]). Thus most of the total computation time is spent in ScaLapack routines.

Unfortunately, at the time the present work was performed (August 1998), the SLHPF interface did not work in conjunction with the particular HPF compilers available on our Meiko CS-2 platform. Therefore, to give some indication of the expected HPF-ScaLapack performance, Table 2 shows the performance

of the ScaLapack Cholesky factorization routine called from Fortran 77 on the Meiko CS-2. In order to utilize the highly optimized Sunperf library, a different compiler (SUN F77) was used than in the experiments of Section 2.

Since the overhead of the SLHPF wrapper routines is small (see Blackford et al. [5]), we expect to achieve nearly the same performance when calling ScaLapack from HPF via the SLHPF interface. Note that the observed efficiencies are comparable with those obtained using HPF with Blas routines optimized with Atlas (see Fig. 6).

Table 2. Performance and efficiency of the Cholesky factorization on $p = 1, 2, \ldots, 16$ processors of a Meiko CS-2 using the Sunperf library and ScaLapack. The matrix size was $n = 1000$.

Processor				Performance [Mflop/s]		
p	Grid	Time	Speedup	Peak	Observed	Efficiency
1	1×1	11.9 s	—	50	28	56 %
2	2×1	6.3 s	1.9	100	53	53 %
4	2×2	3.8 s	3.1	200	88	44 %
8	4×2	1.9 s	6.3	400	175	44 %
16	8×2	1.3 s	9.2	800	256	32 %

Lorenzo et al. [13] developed another interface from HPF to ScaLapack. In this interface, to call the PBlas routine `Pdgemm`, for example, an HPF wrapper routine `HPF_Pdgemm` is called. This sets up the array descriptors of the Blacs (Basic Linear Algebra Communication Subprograms) using the HPF library function `HPF_DISTRIBUTION`, broadcasts this information to all processors, and then calls the PBlas routine `Pdgemm` as an `EXTRINSIC (F77_LOCAL)` routine.

It should be noted that these interfaces from HPF to parallel libraries should also be useful for converting existing *sequential* numerical libraries to HPF. In many cases it should suffice to replace calls to sequential Blas or Lapack routines in the Fortran 77 code by calls to PBlas or ScaLapack routines in the HPF version.

There are, however, two possible sources of difficulty in interfacing HPF to parallel libraries:

- If the HPF compiler is not based on MPI, the initialization of the message passing library is difficult (Blackford et al. [4]). In this case, PVM may be used. PVM computation nodes may also be started and initialized during runtime, independent of the HPF runtime system.
- If the backend compiler is not the standard compiler of a given platform, incompatibilities with the optimized Blas libraries may occur.

3.2 ScaLAPACK Implemented in an HPF Runtime System

The public domain HPF compilation system ADAPTOR (Brandes and Greco [6]) contains an interface to SCALAPACK. The interface is implemented directly in the HPF runtime system using the language C. This approach offers the following advantages.

- A more flexible redistribution strategy is possible.
- Subsections of arrays can be used without creating temporary arrays.
- Overhead resulting from the conversion of the array descriptors from HPF to PBLAS is reduced.

However, portability is reduced due to the proprietary implementation.

4 Conclusions

It has been shown that existing numerical high performance libraries can be integrated into HPF code in a portable and efficient way, using HPF language features only. In particular, it is possible to implement parallel BLAS on top of the sequential BLAS. This guarantees high local performance in HPF programs and yields significant performance improvements compared to pure Fortran implementations which do not take advantage of existing software packages and libraries.

Most high level linear algebra libraries are based on the BLAS. By invoking a parallelized BLAS version implemented along the lines suggested in this paper, these libraries can also be utilized within HPF programs.

Acknowledgments. We would like to thank John Merlin (VCPC, Vienna) for his helpful comments, his experienced interpretation of the HPF standard, and for proofreading the paper.

References

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 *User's Guide*, 2nd ed. SIAM Press, Philadelphia 1995.
2. J. Bilmes, K. Asanovic, C.-W. Chin, J. Demmel, *Optimizing Matrix Multiply using* PHIPAC*: a Portable, High-Performance, ANSI C Coding Methodology*, Proceedings of the International Conference on Supercomputing, ACM, Vienna, Austria, 1997, pp. 340–347.
3. J. Bilmes, K. Asanovic, J. Demmel, D. Lam, C.-W. Chin, *Optimizing Matrix Multiply using* PHIPAC*: a Portable, High-Performance, ANSI C Coding Methodology*, Technical report, LAPACK Working Note 111, 1996.
4. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley: SCALAPACK *Users' Guide*, SIAM Press, Philadelphia 1997.

5. L. S. Blackford, J. J. Dongarra, C. A. Papadopoulos, and R. C. Whaley: *Installation Guide and Design of the HPF 1.1 Interface to* ScaLapack, *SLHPF*, LAPACK Working Note 137, University of Tennessee 1998.
6. T. Brandes and D. Greco: *Realization of an HPF Interface to* ScaLapack *with Redistributions*. High-Performance Computing and Networking. International Conference and Exhibition, Springer-Verlag, Berlin Heidelberg New York Tokyo 1996, pp. 834–839.
7. J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson: *An Extended Set of* Blas, ACM Trans. Math. Software 14 (1988), pp. 18–32.
8. J. J. Dongarra, J. Du Croz, S. Hammarling, I. Duff: *A Set of Level 3* Blas, ACM Trans. Math. Software 16 (1990), pp. 1–17.
9. H. J. Ehold, W. N. Gansterer, and C. W. Ueberhuber: *HPF—State of the Art*, Technical Report AURORA TR1998-01, European Centre for Parallel Computing at Vienna, Vienna 1998.
10. R. van de Geijn: *Using* PLapack*: Parallel Linear Algebra Package*, MIT Press 1997.
11. High Performance Fortran Forum: *High Performance Fortran Language Specification Version 2.0*, 1997.
    URL: `www.crpc.rice.edu/HPFF/hpf2/` or
    `www.vcpc.unvie.ac.at/information/mirror/HPFF/hpf2/`.
12. C. L. Lawson, R. J. Hanson, D. Kincaid, F. T. Krogh: Blas *for Fortran Usage*, ACM Trans. Math. Software 5 (1979), pp. 63–74.
13. P. A. R. Lorenzo, A. Müller, Y. Murakami, and B. J. N. Wylie: *HPF Interfacing to* ScaLapack, Third International Workshop PARA '96, Springer-Verlag, Berlin Heidelberg New York Tokyo 1996, pp. 457–466.
14. C. W. Ueberhuber: *Numerical Computation*, Springer-Verlag, Berlin Heidelberg New York Tokyo 1997.
15. R. C. Whaley, J. J. Dongarra, *Automatically Tuned Linear Algebra Software*, Technical Report, Lapack Working Note 131, 1997.