Utilization of Ready-Made Software in HPF Programs

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

[1] VCPC, European Centre for Parallel Computing at Vienna,
Liechtensteinstrasse 22, A-1090 Vienna, Austria
E-Mail: ehold@vcpc.univie.ac.at

[2] Institute for Applied and Numerical Mathematics,
Vienna University of Technology,
Wiedner Hauptstrasse 8-10, A-1040 Vienna, Austria
E-Mail: ganst@aurora.tuwien.ac.at, christof@uranus.tuwien.ac.at

[3] Institute for Physical and Theoretical Chemistry,
Vienna University of Technology,
Getreidemarkt 9/158S, A-1060 Vienna, Austria
E-Mail: dieter@titania.tuwien.ac.at

The work described in this report was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

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 [7]) 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 advanced features of HPF efficiently. There is, however, another crucial aspect, which is often ignored. There is not enough support for integrating highly optimized library routines into HPF code.

Much effort has been spent on highly efficient implementations of the Basic Linear Algebra Subroutines (Blas [4]) and on numerical libraries for dense linear algebra, which use the Blas as building blocks. Important examples are Lapack [1] as the standard sequential library for dense (or banded) linear algebra methods and parallel libraries such as ScaLapack [2] or PLapack [6].

The integration of existing 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, Gansterer, and Ueberhuber [5]).

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

The basic concept provided by HPF for integrating procedures from other programming languages or from other programming models is the EXTRINSIC mechanism (HPF Forum [7]). Based on this concept, various different ways at different levels of abstraction for combining HPF with existing library routines will be outlined in this paper.

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 level 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 methods and attempts to integrate parallel library routines into HPF. In both cases experiments have been performed, which substantiate the considerable performance gains of numerical applications using the techniques suggested.

The remarkable benefit of the approaches described in this paper is that they are fully *portable*. In contrast to other methods they are independent of compiler or hardware specifics (Section 3). They only use language features of "core HPF" from the HPF standard and a few of the Approved Extensions (HPF Forum [7]) (the extrinsic kinds HPF_LOCAL and F77_SERIAL), which are supported by most HPF compilers.

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 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. A standard Fortran implementation of the BLAS (LIBBLAS), compiled with the *pgf77* Fortran 77 compiler from PGI had to be used in the experiments. Further significant performance improvements are expected when highly optimized BLAS can be 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. A new routine called `par_dgemm` (*parallel dgemm*) has been 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/*gemm`.

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 the computations involved.

2.1.1 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 for parallelization are mentioned here, depending on which index varies in 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 a column of A and of a row of B. In order to localize all the computations one column of A has to be replicated in the direction of the rows of C and one row of B has to be replicated in the direction of the columns of C.

i$*$ variant: The two inner loops compute a row of C by multiplying a row of A with B. In this case it is usually most efficient to parallelize over 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 A with a column of B. In this case it is usually most efficient to parallelize over 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 for two-dimensional data distributions, 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.

In the context of linear algebra algorithms normally two-dimensional data distributions are used (for load-balancing reasons). 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.

2.1.2 Implementation

For performance reasons, a *blocked* version of matrix-matrix multiplication (Ueberhuber [9]) was implemented. Two levels of wrapper routines are involved. The programmer calls the 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 the block column of A and for the 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. It may cause a significant amount of communication, but it has the big benefit of making the procedure fully independent of the prior distributions of A and B. All the necessary communication, which is entirely contained in the copying operation, is restricted to the outermost loop.

2.1.3 Experiments

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

A considerable performance improvement of `par_dgemm` over MATMUL is evident (roughly by a factor of 2). This holds for other distribution scenarios as well. Moreover, for large matrix orders n the efficiency of `par_dgemm` improves, whereas the efficiency of the vendor supplied MATMUL routine decreases. Despite its speedup effect, the efficiency of `par_dgemm` is still at a disappointingly low level. Unfortunately it was not possible to use the highly optimized version of the BLAS as part of the SUN performance library (SUNPERF) because the HPF compiler currently available to the authors on the test machine does not support it. If a highly optimized BLAS library can be utilized instead of the standard version a significant performance improvement is to be expected.

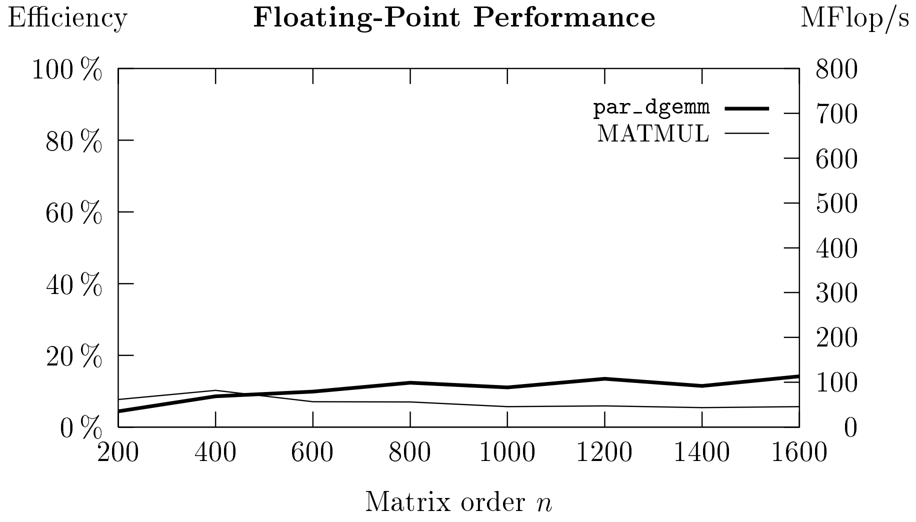

Figure 1: Matrix multiplication on 16 processors of a Meiko CS-2.

In order to evaluate parallel speedup (see Fig. 2), the runtime of the *best* available sequential version (which is a single call to `BLAS/dgemm`) was used as a reference value. Parallel speedup for `par_dgemm` is much better than for parallel MATMUL. The routine shows a satisfactory scaling behavior. The decreasing speedup for $p = 16$ processors indicates that the problem size $n = 1000$ becomes too small for an efficient parallelization.

2.2 Cholesky Factorization

Even if the HPF compiler is doing a good job with organizing data distribution and communication between processors, the performance achieved can be disappointing due to bad node performance. Comparisons with the efficiency of a highly optimized library based on the BLAS, show that the sequential BLAS ver-

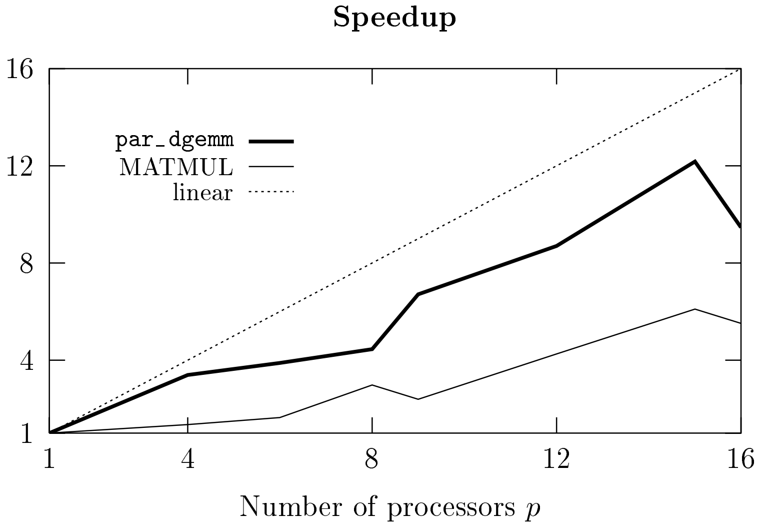

Figure 2: Matrix multiplication on p processors for $n = 1000$.

sion is up to 12 times faster than the fastest parallel HPF version on one processor and still 1.3 times faster than HPF on 16 processors (see Table 1).

Table 1: Cholesky factorization of a matrix of order 1000, using the vendor optimized `LAPACK/dpotrf` or pure HPF on a Meiko CS-2.

p	1	2	4	8	16
BLAS	**9 s**				
HPF	113 s	57 s	29 s	18 s	12 s

One method to improve 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 $A = LL^\top$ of a real symmetric positive definite matrix, exhibits better locality of reference and is therefore to be preferred over the unblocked version on modern computer architectures (Anderson et al. [1], Ueberhuber [9]). 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. 3).

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

This step requires the following three tasks.

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

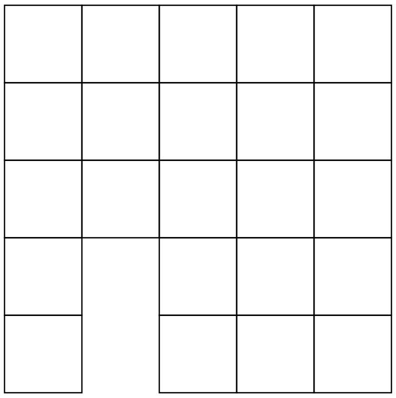

Figure 3: Update areas of blocked Cholesky factorization.

In the HPF version the first task is done sequentially on one processor by using the HPF extrinsic kind F77_SERIAL, but the following steps (`BLAS/*trsm` and `BLAS/*gemm`) are parallelized. For simplicity it is assumed that the block size *hb* divides the dimension of the matrix n evenly, and that $n = lda$ (leading dimension of matrix A). The distribution chosen for the symmetric matrix A is (CYCLIC(*hb*),∗). The one-dimensional distribution was chosen to avoid communications in rows.

2.2.1 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 there is not just one single call to `BLAS/*trsm` but there is one on each processor that owns local data of the block A_{21} (see Fig. 4). Hence in the example shown in Fig. 4 (two processors $P1$ and $P2$) there are two calls to `BLAS/*trsm`. 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.

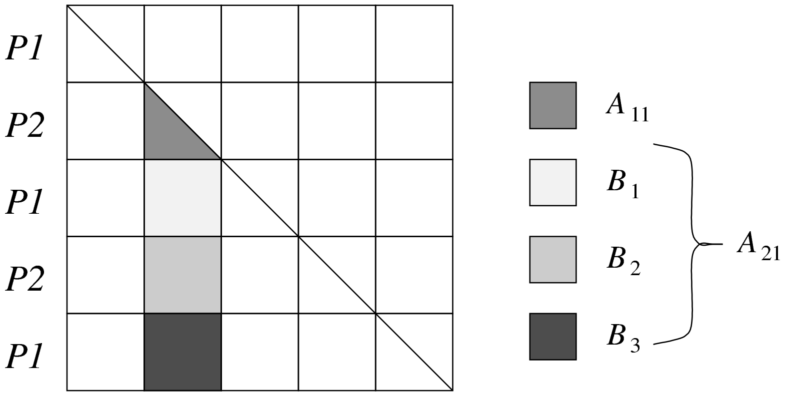

Figure 4: Local blocks in parallel version.

The HPF_LOCAL version calls `BLAS/*trsm` 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, which are portable between all HPF compilers that support the core of HPF 2.0, INHERIT and TEMPLATES, were investigated.

1. The natural way to do this in HPF is by using the INHERIT directive in the interface and the declaration of the HPF_LOCAL routine. INHERIT does exactly what is needed, it tells the compiler that the distribution of an array and hence also subsections of this array have the same distribution in the subroutine as in the caller 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. The `dim` parameter gives the size of the first dimension of the array in the calling routine. This value is used to declare an equal-sized TEMPLATE in the interface of the callee that is equally distributed as the starting array. The parameter `row` 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.

2.2.2 Using F77_LOCAL

The extrinsic kind F77_LOCAL refers to procedures which are implemented in Fortran 77 and in the local programming model. Thus it seems that F77_LOCAL were the natural choice for combining HPF with the BLAS. Since Fortran 77 does not provide the intrinsic function SIZE, 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 [7]), hence the compiler vendors are allowed to implement it in any way they want. For the HPF compiler on the Meiko CS-2 the vendor recommended not to use INHERIT together with F77_LOCAL, hence no performance results can be given for this version.

2.2.3 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. 5 shows just one "HPF+BLAS" curve.

The sequential routine `LAPACK/dpotrf` was benchmarked just on one processor. The efficiency value for p processors was determined by dividing the single-processor value by p.

The efficiency of the HPF version combined with the BLAS is decreasing with increasing number of processors. The reason for this phenomenon is that the number of blocks on each processor, and hence the amount of computation, is becoming smaller and smaller (due to the fixed problem size). On one processor the superiority of the (not vendor optimized) BLAS over the produced HPF code can be seen.

3 Calling Parallel Libraries from HPF

As an alternative to the previous section, extrinsic *parallel* library routines can be invoked for *distributed* computations. In this case, communication is also performed within the extrinsic procedure. HPF is used as a framework for conveniently distributing data and for organizing high-level parallelism. It should be noted that the HPF compilation system has to be compatible with the mes-

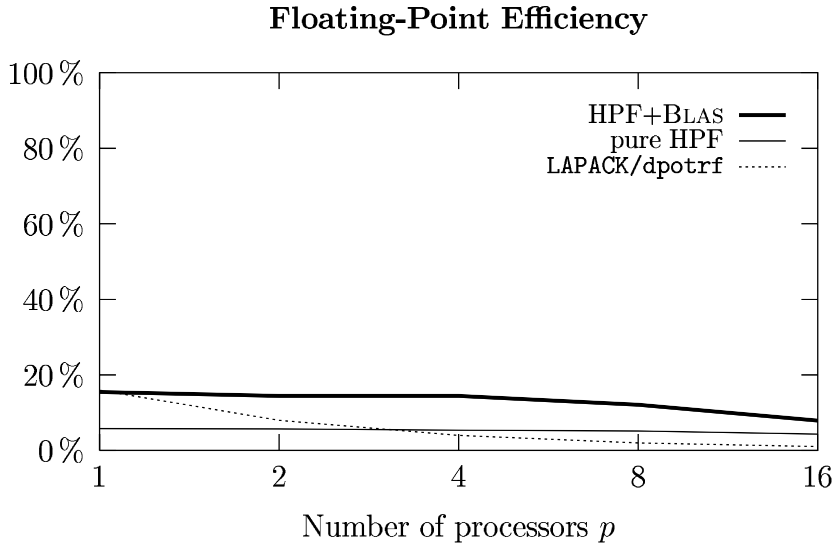

Figure 5: Efficiency of the Cholesky factorization using a matrix of order 2000.

sage passing library used by the software package and possibly also with (usually precompiled) computational libraries (for example, the BLAS).

3.1 Parallel BLAS

On a higher level of abstraction (compared to the techniques suggested in Section 2) parallel versions of the BLAS can be integrated into HPF. One approach for implementing parallel BLAS on top of sequential BLAS merely using HPF language features was shown in Section 2. As an alternative, the PBLAS (as defined in the SCALAPACK project; Blackford et al. [2]) can be used directly.

It should be emphasized that techniques for calling parallel BLAS from HPF are also useful when trying to parallelize existing *sequential* numerical libraries. In many cases it would suffice to replace calls to sequential BLAS by calls to parallel BLAS to get a parallel code.

3.1.1 PBLAS

Calling distributed BLAS as defined in the SCALAPACK project mainly involves the issue of handing over the (possibly incompatible) distribution information from HPF to the library routines.

The representation and internal handling of this distribution information is not standardized and depends on the particular compiler. Compiler vendors sometimes actually prefer not to publish this kind of information so that customers do not rely on a certain format. Vendors want to retain flexibility to change these internal details whenever required.

Lorenzo et al. [8] showed that it is possible to implement an interface to the PBLAS with standard HPF (and the extrinsic kind F77_LOCAL). A wrapper routine (e. g., HPF_PDGEMM) sets up the array descriptors of the BLACS (Basic Linear Algebra Communication Subprograms) by using the HPF library function HPF_DISTRIBUTION, spreads this information to all processors, and calls then the PBLAS routine PDGEMM as F77_LOCAL.

3.2 ScaLAPACK and EXTRINSIC Functions

A prototypical port to HPF is included in the SCALAPACK package. This interface from HPF to SCALAPACK uses three layers of wrapper routines: (i) global HPF routines, (ii) HPF_LOCAL routines, and (iii) strict Fortran77, taking local assumed-size arrays as arguments.

Among others LU factorization and Cholesky factorization are the first available prototypes in the HPF wrapper library. Unfortunately, these wrapper routines would not run in most current execution environments.

The wrapper routines do not have a significant influence on the execution time. Thus the main contribution to the total computation time is spent in SCALAPACK routines.

Table 2: Execution time of the Cholesky factorization on a Meiko CS-2 using the SUNPERF library and SCALAPACK. The matrix size was 1000.

Number of Processors	Process Grid	Execution Time	Speedup	Peak Performance	Achieved Performance
1	1×1	11.9 s	1.0	50 Mflop/s	28 Mflop/s
2	2×1	6.3 s	1.9	100 Mflop/s	53 Mflop/s
4	2×2	3.8 s	3.1	200 Mflop/s	88 Mflop/s
8	4×2	1.9 s	6.1	400 Mflop/s	175 Mflop/s
16	8×2	1.3 s	8.9	800 Mflop/s	256 Mflop/s

Another interface from HPF to SCALAPACK was written by Lorenzo et al. [8]. This interface is similar to the one described in Section 3.1.1.

3.3 ScaLAPACK Implemented in HPF Runtime System

The public domain HPF compilation system ADAPTOR (Brandes and Greco [3]) comprises an interface to SCALAPACK. ADAPTOR consists of a source-to-source

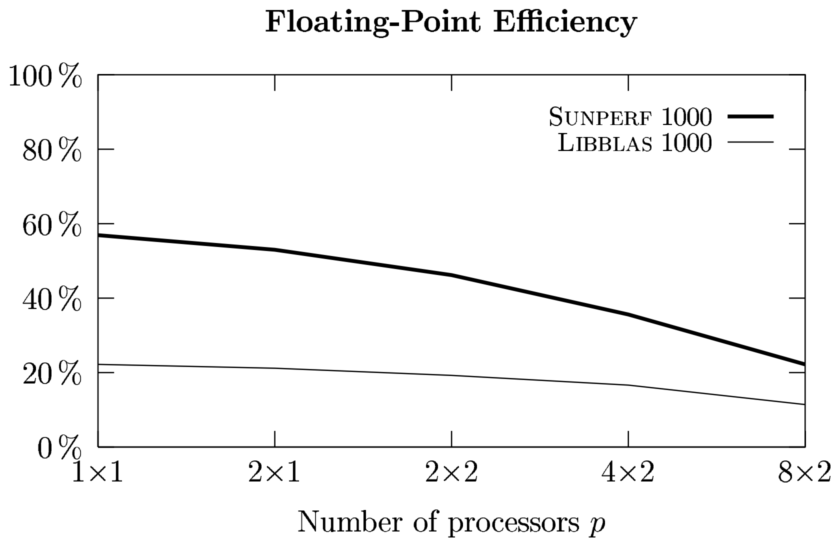

Figure 6: Efficiency of the Cholesky factorization on a Meiko CS-2 using SCALAPACK. The matrix size was 1000.

translator (FADAPT) and a runtime library (DALIB). HPF programs are translated into SPMD message passing programs, which use the runtime library for doing the message passing.

The interface is implemented in the HPF runtime system directly by use of 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.

On the other hand some problems arise.

- Portability is reduced due to the proprietary implementation.
- This interfacing does not work using PVM, because the virtual machine of PVM can only be started from either the HPF runtime system or from the BLACS library used by SCALAPACK. Thus MPI has to be used.

3.4 Performance of HPF and ScaLAPACK

Fig. 6 and Table 2 show that the achieveable performance is very high, provided an optimized BLAS library is available. Note, the compiler system was different from the one used in Section 2.

Two versions of BLAS libraries have been compared. The highly optimized SUNPERF library, and the moderately optimized LIBBLAS library. The faster version, SUNPERF, shows less speedup only because the time required for local computation is relatively small compared to communication time.

The wrapper routines do not require much computation time, so the full performance of SCALAPACK can be utilized from within HPF (as soon as an appropriate interface is available). However, in two areas difficulties may occur.

- If the HPF compiler is not based on MPI, the initialization of the message passing library is difficult (Blackford et al. [2]). 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 on a given platform, incompatibilities with the optimized BLAS libraries may occur (see Section 2.1).

If standard SCALAPACK routines have to be used in a parallel implementation, HPF in a suitable environment is definitely a good choice. If modified libraries have to be parallelized, the interface to PBLAS may be used. Some of the difficulties occuring in specific implementations may be overcome by applying the methods of Section 2.

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.

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] 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.

[3] 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.

[4] J. J. Dongarra, J. J. Du Croz, I. S. Duff, and S. J. Hammarling: *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Software 16 (1990), pp. 1–17.

[5] 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.

[6] R. van de Geijn: *Using* PLAPACK*: Parallel Linear Algebra Package*, MIT Press 1997.

[7] 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/`.

[8] 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.

[9] C. W. Ueberhuber: *Numerical Computation*, Springer-Verlag, Berlin Heidelberg New York Tokyo 1997.