Optimizing Local Performance in HPF

Harald J. Ehold* Wilfried N. Gansterer* Dieter F. Kvasnicka†
Christoph W. Ueberhuber*

September 3, 2001

Abstract

High Performance Fortran (HPF) was created to simplify high-level programming on parallel computers. The inventors of HPF strove for an easy-to-use language which was intended to enable portability *and* efficiency. However, up until now the desired efficiency has not been reached. On the contrary, HPF programs are notorious for their poor performance.

This paper provides a rehabilitation of HPF. It is demonstrated how currently available HPF constructs can be utilized to solve sizeable numerical problems efficiently. The method suggested utilizes HPF's EXTRINSIC mechanism to integrate existing numerical *single processor software* for computationally expensive kernels into HPF programs.

By using the technique described in this paper, the empirical efficiency, i. e., the ratio of the empirical floating-point performance to the theoretical peak performance, can be raised to 50 % and more. Even on message passing machines with slow communication networks, such as PC clusters (Beowulf clusters) using a 100 Mbit/s Ethernet interconnection, highly satisfactory empirical efficiency results. The performance achieved is even competitive with that of well established numerical libraries based on MPI.

In contrast to earlier approaches for utilizing existing numerical software in HPF programs, the method presented here uses only HPF features and is therefore portable.

Keywords: High Performance Fortran, HPF, Numerical Linear Algebra, Parallel Computing, Numerical Libraries.

*Institute for Applied and Numerical Mathematics, Vienna University of Technology, Austria.
`Harald.Ehold@nokia.com`, `{ganst,christoph}@aurora.anum.tuwien.ac.at`
†Institute for Physical and Theoretical Chemistry, Vienna University of Technology, Austria.
`dieter.kvasnicka@tuwien.ac.at`

1 Introduction

High Performance Fortran (HPF [19]) is a programming language which provides high-level support for the development of parallel programs. It has been designed primarily for regular, data-parallel applications. One of the central goals of HPF is to combine *high performance* with *portability* across a wide range of (distributed memory) parallel computers.

Since the introduction of HPF 1.0 [18] in 1992 by the High Performance Fortran Forum, two important trends have influenced its further development.

- On the one hand, compiler producers were reluctant to invest a lot of effort into the development of high quality HPF compilers, because it was unclear how quickly good compilers for Fortran 90/95, which forms the basis of HPF, would become available. Moreover, it was somewhat controversial how successful HPF, and more generally, automatic parallelization tools for distributed memory parallel computers, could ever be.
- Users, on the other hand, soon started to demand more advanced features to deal with a broader range of applications than just the regular data-parallel ones. In particular, support for irregular distributions and for task parallelism was investigated (see, for instance, Chapman et al. [11], Mehrotra et al. [22], Das et al. [12], Saltz et al. [27], Benkner [4, 5]).

As a consequence of the first trend, several central features of HPF 1.0, which are hard to implement efficiently, were taken out of the core language and given the status of "Approved Extensions". At the same time, according to the second trend, new language constructs, mainly directed towards irregular applications, were developed and incorporated into the Approved Extensions. The result of these activities is the current version of the language definition, HPF 2.0 [19].

HPF is a conceptually very elegant and attractive approach to high-level parallel programming. In particular, it provides very convenient ways for specifying data distribution and for expressing data-parallelism. The development of parallel code using HPF is easier and requires less coding effort than explicitly programming message-passing, for example, using MPI [23]. Nevertheless, HPF has not yet become widely accepted by users for several reasons (Ehold et al. [16]).

- It took a long time for the language standard HPF 2.0 [19] to be fully supported by commercial compilers. Only recently have mature compilers become available. Unfortunately, several companies have suspended their development of HPF compilers because the market has been assessed as being too small.
- The few HPF compilers which are available are not able to deliver acceptable performance. In fact, in many cases the performance of HPF codes is so inferior to explicit message-passing programming that HPF can not be considered a competitive alternative despite its obvious advantages in terms of code development, flexibility, maintenance, portability, and debugging.

In this paper a concept for achieving high performance using HPF is introduced, which opens up new perspectives. The central idea is to utilize existing high quality (sequential) software for *local* computations.

1.1 Related Work

Previous work in this area has concentrated on interfacing HPF to extrinsic *parallel* library routines from numerical software packages such as ScaLapack [7]. In this approach the extrinsic parallel library routine performs communication as well as computation and the main issue is how to efficiently pass distribution information from HPF to the extrinsic parallel routine. In contrast, the focus of our method is on interfacing HPF to *sequential* library routines. HPF is used as a framework for conveniently distributing data and for organizing high-level parallelism.

Blackford et al. [8] have developed SLHPF, an interface from HPF to ScaLapack. 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 containing calls to the Blacs and to ScaLapack. The current version of the SLHPF interface contains HPF wrapper routines for ScaLapack's LU and Cholesky factorization and wrapper routines for the Level 3 PBlas routines `PBLAS/pdgemm` and `PBLAS/pdtrsm`, among others. The wrapper routines do not have a significant influence on the overall execution time (see Blackford et al. [8]). Usually most of the total computation time is spent in ScaLapack routines, and the performance of HPF routines which call ScaLapack routines via the SLHPF interface is similar to the performance of directly called ScaLapack routines. However, if the data distribution of the HPF code is not compatible with ScaLapack, the matrices have to be redistributed at the interface. In this case, performance may deteriorate significantly.

Lorenzo et al. [21] developed another prototypical interface from HPF to ScaLapack for an earlier version of the PGI HPF compiler.

The public domain HPF compilation system Adaptor (Brandes and Greco [9]) also contains an interface to ScaLapack. This interface is directly built into the runtime system of the Adaptor system, which makes it possible to bypass the EXTRINSIC mechanism of HPF. In contrast to these approaches, which require a rather complicated special interface to the parallel library routines, the method described in this paper only requires the EXTRINSIC interface which is part of the HPF standard.

From a more general point of view, there are obviously numerous other approaches in the area of automatic parallelization, such as OpenMP [10, 25]. However, it is not our intention to discuss the state-of-the-art or the future potential of automatic parallelization as a whole. Instead, we focus on HPF as a specific example. We show how highly optimized library routines can be made accessible to HPF and illustrate that significant performance improvements are possible by supporting an automatic parallelization tool like HPF with such external building blocks.

1.2 Synopsis

In this paper it is shown that HPF programs can meet high performance expectations when combined with existing optimized library routines. It is also shown that HPF programs utilizing *sequential* library routines are preferable to HPF programs utilizing *parallel* libraries in many cases. This is not only due to the fact that the HPF constructs required when utilizing sequential routines are already available in HPF compilers and that, therefore, the resulting code is highly portable; it is also due to the fact that this approach is less sensitive to the data distribution in terms of performance.

In Section 2 our basic methodology for integrating sequential library routines into

HPF programs is introduced. In Section 3 it is shown how this concept can be applied successfully to some very important operations in high performance scientific computing: general and symmetric matrix operations, Cholesky factorization, and two-dimensional FFT. In Section 4 the results of numerical experiments carried out on various parallel computers are summarized.

2 HPF and Numerical Libraries

A considerable amount of expertise has been incorporated into high quality software like the BLAS [13, 14, 20], into packages which use the BLAS as building blocks like LAPACK [2], the standard sequential package for dense or banded matrices, and into parallel packages such as SCALAPACK [7] or PLAPACK [29]. These sequential and parallel libraries can be seen as the result of several decades of combined effort in the field of numerical computing. It seems useful to make these optimized libraries accessible to an HPF program and to use their routines as highly efficient building blocks. In this paper we discuss a basic concept how this can be achieved and illustrate its advantages with a few examples. Ultimately, the goal would be to create a library of parallel routines based on this concept, so that a user could access all the functionality *and the efficiency* provided by the BLAS and LAPACK from any HPF program without having to know the implementation details of these library routines.

2.1 Standard Numerical Operations in HPF

Experience shows that native[1] HPF code executing standard numerical operations is often not able to utilize the potential floating-point power of a given computer system efficiently.

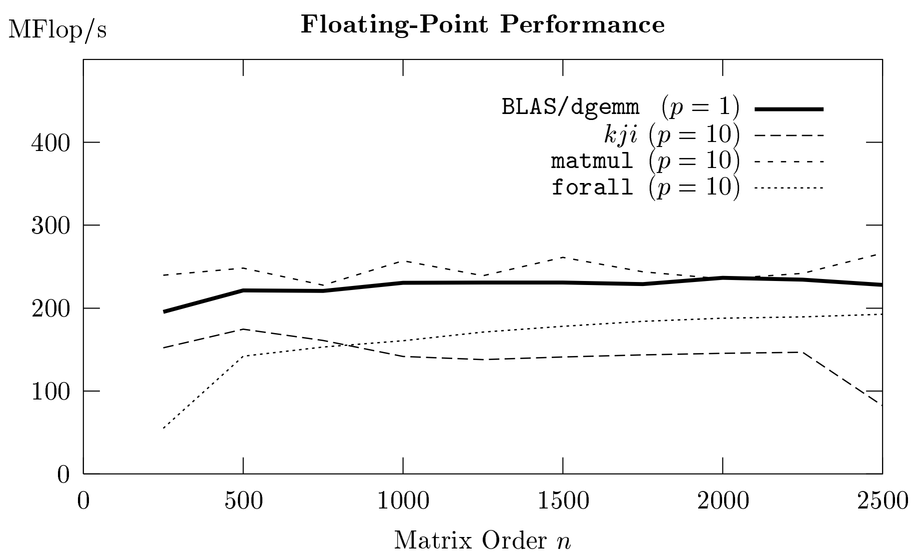

Figure 1: Matrix-matrix multiplication on $p = 1, 10$ processors of a Beowulf Cluster (see Section 4.1) whose peak performance is $p \times 350$ Mflop/s.

[1] HPF routines which do not use the EXTRINSIC mechanism are called *native* in this paper.

Take, for example, the floating-point performance of two different native HPF implementations of matrix-matrix multiplication (labelled "*kji*", "`forall`") and the intrinsic procedure `matmul` on 10 processors ($p = 10$) of a PC cluster compared to a *sequential* `BLAS/dgemm` implementation on *one* single processor ($p = 1$) as depicted in Fig. 1. This diagram shows that the *absolute* performance of the sequential BLAS routine *on one processor* is equal to or better than that of native HPF codes running *on ten processors*.

Another example of this poor performance can be found in comparisons of various implementations of Cholesky factorization (see Table 1). Here it turned out that the *sequential* LAPACK routine `LAPACK/dpotrf` is 1.5 times faster than a native HPF implementation with nearly linear parallel speed-up *on eight processors*. However, a newly developed parallel HPF routine utilizing sequentially optimized BLAS routines (labelled "`par_dpotrf`"), which will be discussed in Section 3.3, is even faster than the corresponding highly tuned SCALAPACK routine and shows a similar speed-up.

Table 1: Cholesky factorization of a symmetric 3000×3000 matrix on a Beowulf cluster using (i) an LAPACK routine, (ii) a native HPF routine, (iii) a SCALAPACK routine, and (iv) a new routine using HPF and the BLAS (see Section 3.3).

Number of processors p	1	2	4	8
`LAPACK/dpotrf`	**43.5 s**			
Native HPF	472.7 s	214.1 s	105.9 s	63.8 s
`SCALAPACK/pdpotrf`	83.1 s	47.9 s	27.6 s	19.2 s
`par_dpotrf`	48.6 s	34.2 s	25.3 s	18.4 s

It should be emphasized that in both examples the best HPF code was compiled with the best available HPF compiler and that an optimizing Fortran compiler was used for the sequential (single processor) version of the Cholesky factorization. Among the users of numerical software it is well known that the best sequential performance can be achieved by utilizing optimized BLAS routines. It is to be expected that the parallel runtimes of the HPF codes will be reduced as better compilers, which are able to optimize the communication, become available. However, for the sequential *local* computations the compilers won't be able to achieve the performance of optimized libraries like the BLAS.

Utilization of available high quality software is desirable not only in terms of performance but also for a very practical reason: In many cases the programming and debugging effort can be reduced considerably by using existing library routines as building blocks instead of coding entire algorithms in HPF directly. This is true, in particular, for applications which involve computationally intensive mathematical operations. Thus, when HPF is used for parallelization, the necessity of combining routines from numerical libraries with HPF code arises quite naturally.

2.2 Optimizing Local Performance

Even if a certain problem is well suited for HPF implementation, the performance of the code relies heavily on the maturity of HPF compilers, which cannot yet be taken for granted (Ehold et al. [16]). But even if an HPF compiler does a good job organizing data distribution and communication between processors, the actual performance achieved can still be disappointingly low due to inefficiencies in the local computations (see Section 2.1). According to our experience, one of the key issues involved in improving overall parallel performance of HPF codes is the optimization of local (sequential) performance.

Sequential optimization typically involves restructuring the code, for example, by increasing the fraction of Level 3 (matrix-matrix) operations and by using highly optimized Level 3 BLAS routines (or other easily accessible high performance library routines) wherever possible.

Highly optimized BLAS implementations are provided by many vendors of computer systems. Moreover, efficient BLAS implementations can be generated individually using tools like PHIPAC (Bilmes et al. [6]) or ATLAS (Whaley and Dongarra [31], Whaley et al. [32]). These code generators automatically find a very good set of hardware dependent parameters for the Level 3 BLAS, and can thus produce highly efficient code for these routines.

Clearly, HPF users should be able to utilize such resources. Highly optimized sequential software helps to maximize the local performance of parallel programs. This, in turn, improves overall parallel performance, provided the algorithm is not communication bound on a certain computer. Moreover, the use of widely available optimized routines (like the BLAS) for local computations ensures *performance portability.*

The following sections of this paper describe how routines from existing sequential numerical packages ("EXTRINSIC routines" in HPF terminology) can be integrated into HPF programs for performing *local* computations. The HPF compiler, supported by user directives, remains responsible for handling and organizing parallelism.

The basic concept and main ideas presented in this paper are applicable to many data-parallel mathematical operations (which come mostly, but not exclusively, from Linear Algebra). In this paper, only a few representative operations are discussed in detail. In the future it would be desirable to generate a complete library of parallel routines for important basic vector and matrix operations which can be called from HPF and achieve high performance by utilizing optimized sequential routines for local computations. Users would not have to know the implementation and parallelization details of these routines. They could use them as "black boxes" within parallel HPF programs, just like the BLAS and LAPACK routines are used as "black boxes" within sequential Fortran programs.

2.3 HPF Features Needed for High Local Performance

The basic facility for integrating procedures from other programming languages or models into HPF code is the EXTRINSIC mechanism (HPF 2.0 [19]). The concept for achieving high local performance presented in this paper is based on this mechanism. Portability is ensured because this concept only relies on standard HPF features and, additionally, on very few specific features of the HPF 2.0 Approved Extensions (HPF 2.0 [19]). The required HPF features are

EXTRINSIC(F77_LOCAL) subroutines: The extrinsic kind F77_LOCAL refers to procedures that are implemented in Fortran 77 and in the *local* programming model. Each processor executes their Fortran 77 code on its local data. This suggests that F77_LOCAL should be the natural choice for combining HPF with routines like the BLAS, which use Fortran 77 calling conventions. However, since the Fortran 90 intrinsic function SIZE, which we use to determine the size of *local* arrays, is not supported within an F77_LOCAL subroutine, HPF_LOCAL has to be used, too. F77_LOCAL is not part of the HPF 2.0 standard, but it is included in the Approved Extensions. Its actual implementation may vary to some extent from compiler to compiler.

EXTRINSIC(HPF_LOCAL) subroutines: The extrinsic kind HPF_LOCAL refers to procedures implemented in HPF in the local programming model. Each processor executes their HPF code only on the local portions of distributed data. Subroutines of this kind provide a means of handing arguments over to local BLAS routines.

INHERIT: The INHERIT directive tells the compiler that the distribution of an array should remain identical when this array or a subsection of it is passed to a subroutine. This feature is needed in the routines described in the following. In case INHERIT is not supported by a particular compiler then this functionality could also be achieved using the TEMPLATE directive (see Ehold et al. [15]).

ALIGN: Proper alignment of certain submatrices is one of the most important features of the approach described in this paper. The same data is often required on more than one processor at the same time. In order to account for this, small work arrays are aligned properly (along one dimension) with a given distributed array. Replication of rectangular blocks ensures that all data is resident on all processors where it is required. This alignment of the work arrays in one dimension and replication in the other dimension (see Fig. 2) can be specified with an advanced form of the ALIGN statement: `ALIGN A(j,*) WITH C(j,*)` (replication of `A` along one dimension of `C`). This construct allows to write efficient code which works for many different data distributions (see Section 3).

Some additional HPF features are not essential for achieving high local performance, but significantly reduce the coding effort and improve the readability of the resulting code:

- In some algorithms it is necessary to treat certain matrix elements or subblocks (for example, diagonal blocks) differently from others. When a distributed array is passed to an HPF_LOCAL or to a F77_LOCAL routine then each processor "sees" only the local parts of this array and references the array elements using *local* indices. The routine LOCAL_TO_GLOBAL of the HPF_LOCAL_LIBRARY makes it possible to inquire locally the *global* indices of these array elements. In Section 3.2 this will be used in order to determine whether the local subarrays are above, at, or below the matrix diagonal.
- Some (but not all) of the routines discussed in more detail in this paper make a few assumptions about the initial data distribution (for example, `par_dpotrf` requires a two-dimensional block-cyclic data layout, see Section 3.3) for optimum performance. The inquiry subroutine HPF_DISTRIBUTION is used to determine whether the data is distributed suitably and to obtain the block size of the current distribution, which may then also be used as the algorithmic block size.

3 Examples

To demonstrate the usefulness of the new approach, basic linear algebra operations like matrix multiplication routines as well as higher level algorithms like Cholesky factorization and 2D FFT were implemented. The linear algebra codes utilize Blas routines for local computation. The 2D FFT code calls a sequential FFTpack[2] routine for the local computation of 1D FFTs. As mentioned before, the ultimate goal would be to create a complete library of routines for mathematical operations based on this approach so that users don't have to care about the details of integrating optimized sequential numerical routines into HPF code.

3.1 General Matrix-Matrix Multiplication

An HPF routine called `par_dgemm` (*parallel double precision general matrix-matrix multiplication*) which computes the product $C \in \mathbb{R}^{m\times n}$ of two general matrices $A \in \mathbb{R}^{m\times l}$, $B \in \mathbb{R}^{l\times n}$ has been developed. One of the attractive features of this routine is that the distribution of the matrices involved hardly influences the performance achieved. Internally, the operation is split up into local operations, each of which is performed by calling the *sequential* general matrix-matrix multiplication routine `BLAS/dgemm`.

In general, the multiplication of two distributed matrices involves *non-local* computations. However, by choosing the proper loop order and by replicating the matrices A and B in a blocked fashion, it becomes possible to localize *all* of the computation involved and to distribute the computation evenly among processors.

Loop Ordering. The central operation in matrix-matrix multiplication,

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

appears inside three nested loops $i = 1, 2, \ldots, m$, $j = 1, 2, \ldots, n$, and $k = 1, 2, \ldots, l$. Permuting the order of these loops yields different algorithmic variants (see, for instance, Ueberhuber [28]) which have different parallelization characteristics and consequently require different means for expressing parallelism (see, for instance, Ortega [26]). The variant with k in the outermost loop was chosen for the implementation. The operations of the two inner loops are performed in parallel. This method achieves perfect data locality with a minimum amount of data replication; that is, it requires the smallest amount of replicated data without having to communicate the elements of C. Moreover, varying k in the outer loop is well suited for a two-dimensional data distribution, which is usually preferred for matrix operations (see, for example, ScaLapack [7]).

par_dgemm. For performance reasons, a *blocked* version of matrix-matrix multiplication routines (Ueberhuber [28]) was implemented. Two levels of wrapper routines are involved. The programmer calls the HPF routine `par_dgemm`, which takes the matrices A and B as input parameters and returns the product matrix C.

In `par_dgemm`, work arrays $\tilde{A} \in \mathbb{R}^{m\times b}$ for a block column of A and $\tilde{B} \in \mathbb{R}^{b\times n}$ for a block row of B are aligned properly with C (*partial replication*, see Fig. 2), and then subarrays of A and B are copied into these work arrays. This copying operation in the outermost k-loop adjusts the distribution of the currently required parts of A and B to that of C. It is the only place where inter-processor communication occurs. As a

[2] see `http://www.netlib.no/netlib/fftpack/`

result, the routine becomes independent of the prior distributions of A and B and all communication takes place in the outermost loop.

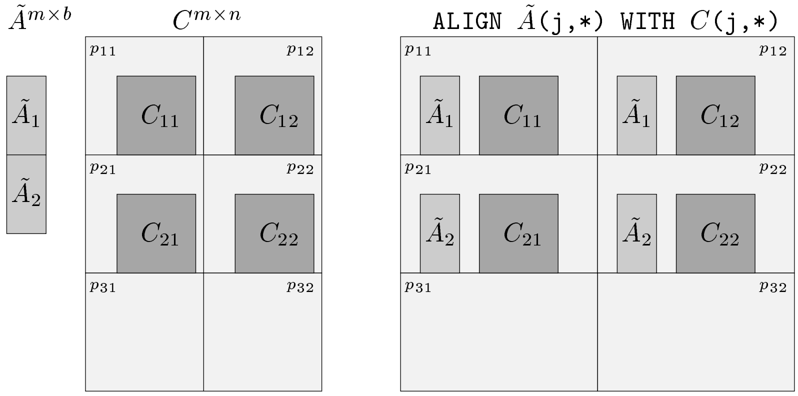

Figure 2: Partial replication of the array $\tilde{A}$ adapted to the distribution of the array C over the processors p_{ij}.

Inside the k-loop the HPF_LOCAL routine `local_dgemm` is called. In addition to the blocks of A and B `local_dgemm` also takes their size (i. e., the block size b of the algorithm) as an argument and computes the outer product of $\tilde{A}$ and $\tilde{B}$. Because of the partial replication, computing this outer product involves only one *local* call to `BLAS/dgemm` on each processor.

3.2 Symmetric Matrix Operations

The following two sections exemplify (i) matrix-matrix multiplication routines for one symmetric and one general matrix resulting in a general matrix (`par_dsymm`: $C = C + AB$, A symmetric) and (ii) a rank-k update of a symmetric matrix (`par_dsyrk`: $C = C + AA^T$).

par_dsymm. The HPF routine `par_dsymm` computes the product of a symmetric matrix $A \in \mathbb{R}^{m\times m}$ and a general matrix $B \in \mathbb{R}^{m\times n}$ resulting in a general matrix $C \in \mathbb{R}^{m\times n}$. The structure of `par_dsymm` is similar to the structure of `par_dgemm`. The main difference is caused by the symmetry of the matrix A. In `par_dgemm` the alignment of the elements of A to C is achieved by copying an entire array section at once. In `par_dsymm` the symmetry of A has to be taken into account, and thus the copy operation is split into three parts (see Fig. 3).

As in the general case of matrix-matrix multiplication, work arrays $\tilde{A} \in \mathbb{R}^{m\times b}$ for a block column of A and $\tilde{B} \in \mathbb{R}^{b\times n}$ for a block row of B are aligned properly with C. Submatrix A_1 is transposed and then copied to $\tilde{A}$. A_2 is copied to $\tilde{A}$ and then transposed and A_3 is copied as it is. The remaining implementation details of `par_dsymm` are identical to those of `par_dgemm`.

The implementation presented here also makes it possible to tune the distribution block size independently of the algorithmic block size. This gives the programmer more flexibility in performance tuning than, for example, the ScaLapack routines do (see Fig. 8).

par_dsyrk. As a counterpart to `BLAS/dsyrk` the HPF routine `par_dsyrk` was developed. The operation performed by this routine is the rank-k update $C := C + AA^T$

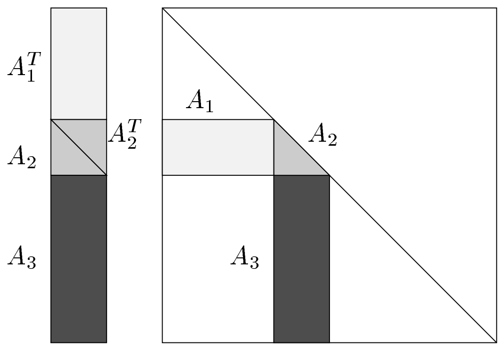

Figure 3: Three copy operations in `par_dsymm`.

with general $A \in \mathbb{R}^{n\times k}$ and symmetric $C \in \mathbb{R}^{n\times n}$. The most natural way of distributing computation of this rank-k update is to communicate only A and to let each "owner" of C do the computation of its local elements. Special care has to be taken with respect to load distribution. When compared to `par_dgemm` and `par_dsymm` it becomes clear that two-dimensional, block-cyclic data distribution leads to the most reasonable load distribution.

New problems arise concerning the shape of local portions of C (see Fig. 4). In contrast to `par_dgemm` and `par_dsymm` the operations on the local portions cannot be combined into a single BLAS call. The solution to this problem is to combine only subblocks of each column block in a call to `BLAS/dgemm` and to compute diagonal blocks by making calls to `BLAS/dsyrk`. The loop over the column blocks is done independently on all the processors (in an HPF_LOCAL routine), so no additional communication is involved. Like in `par_dgemm` and `par_dsymm`, the distribution of A is done outside the HPF_LOCAL routine. If A has too many columns, an additional (higher level) loop is used to split up A into column blocks which are treated sequentially.

To obtain a slightly better performance, operations on the corresponding parts of several local column blocks may be combined into one `BLAS/dgemm` call. However, this variant has not been implemented yet.

For $k \gg n$ other parallelization methods, based on communicating C *and* A, might be advantageous. At the moment, `par_dsyrk` does not implement such techniques.

3.3 Cholesky Factorization

The Cholesky factorization $LL^\top$ of a real symmetric positive definite matrix A serves as another example for applying the techniques for integrating optimized sequential numerical routines into HPF code. Early experiments with native HPF implementations of the Cholesky factorization have shown very poor performance. In fact, the best performance using an HPF code was achieved utilizing all available processors, but it was still slower than `LAPACK/dpotrf` on a single processor of the same machine (cf. Table 1 and Ehold et al. [15]).

Parallelization. *Blocked* Cholesky factorization exhibits better locality of reference and is therefore preferred over the unblocked version on modern computer architectures

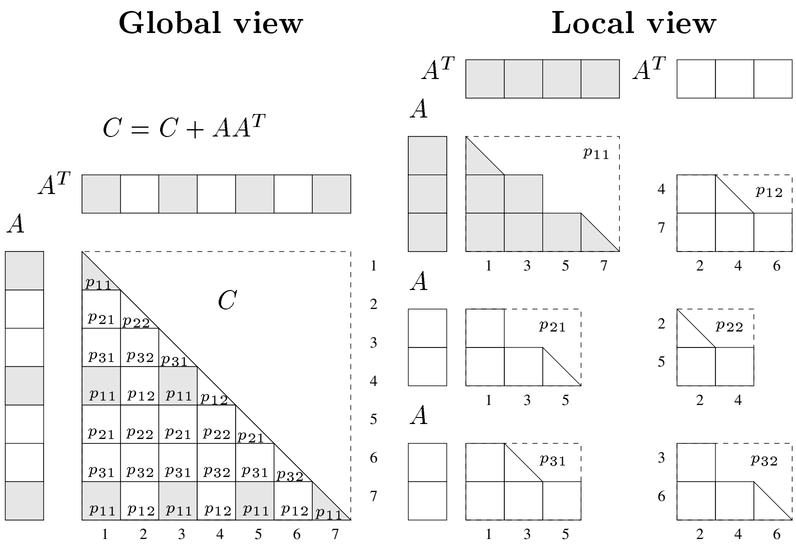

Figure 4: Two-dimensional block-cyclic distribution of matrix C on a 3×2 processor grid p_{ij}. Alignment of matrix A with matrix C.

(Anderson et al. [2], Ueberhuber [28]). Blocked Cholesky factorization

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

consists of the following three tasks (see also Fig. 5).

1. Factorize $A_{11} = L_{11}L_{11}^\top$,
2. Solve the linear system $L_{21}L_{11}^\top = A_{21}$ for L_{21},
3. Update $A_{22}' := A_{22} - L_{21}L_{21}^\top$ and compute the Cholesky factorization of $A_{22}' = L_{22}L_{22}^\top$.

Three versions of this algorithm are (i) left-looking (Task 1 is performed recursively), (ii) right-looking (Task 3 is performed recursively), and (iii) "recursive" Cholesky factorization (Tasks 1 *and* 3 are performed recursively; see Andersen et al. [1]). All routines which were compared in Table 1—ScaLapack, native HPF, and the HPF+Blas routine `par_dpotrf`—implement the right-looking Cholesky factorization.

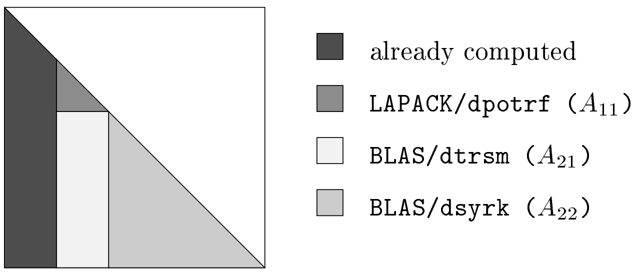

Figure 5: Update areas of blocked Cholesky factorization.

Implementation. The HPF routine `par_dpotrf` for computing the Cholesky factorization in parallel has been implemented. Task 1—which is computationally negligible—is done sequentially by calling `LAPACK/dpotrf` via an extrinsic F77_LOCAL routine on the processor which owns the block A_{11}. `BLAS/dtrsm` is called for the triangular solves in Task 2 on the processors that own parts of the block A_{21}. Since block A_{11} is needed by all participating processors, it is replicated over all processors (like the matrices A and B in `par_dgemm`, see Section 3.1). The matrix update in Task 3 is performed via a call to the HPF wrapper routine `par_dsyrk` (see Section 3.2). Finally, in Task 3 the Cholesky factorization is performed recursively.

`par_dpotrf` imposes a few restrictions on the distribution of the matrix A. If the submatrix A_{11} is of order b, the appropriate distribution of A is (CYCLIC(kb), CYCLIC(lb)) with $k, l \in \mathbb{N}$. This distribution ensures that submatrix A_{11} is local on one processor and that only one processor column owns A_{21}, thus minimizing communication overhead. It should be noted that (i) the code of `par_dpotrf` is very short and easy to understand, and (ii) the parallelization overhead is very small.

3.4 Two-Dimensional FFT

A two-dimensional FFT of a data matrix is usually computed by calculating one-dimensional FFTs for each column and then for each row. This implies a natural way of parallelizing this operation. If the two-dimensional array is distributed columnwise, one-dimensional FFTs along the columns are *local* operations. The two-dimensional array is then transposed in order to make it possible to compute the rowwise one-dimensional FFTs locally as well. The transposition involves all-to-all communication, which leads to considerable communication overhead. Nevertheless, this approach, implemented in the routine `par_fft`, which calls routines from FFTPACK for local computations, is more efficient than using an alternative algorithm based on a static decomposition with parallel one-dimensional FFTs to perform the rowwise transformations (see, for example, Noelle and Jungclaus [24]).

4 Numerical Experiments

In this chapter a short overview of the hardware platforms used in the experiments and a description of the results are given.

4.1 Hardware Platforms

Numerical experiments were carried out on several multiprocessor machines in order to verify the usefulness of the new techniques. The results reported in the following sections were obtained on an IBM SP2 and on a PC cluster.

IBM SP2. The SP2 used in the experiments was a system equipped with 37 processors (POWER2 architecture RS/6000), of which up to 16 at a time were used in the experiments. Communication was provided via the SP2's high performance switch. Theoretical peak performance using p processors was $p \times 267$ Mflop/s. HPF support was provided via Portland Group's HPF compiler *pghpf* version 2.4-4. The local BLAS routines used were part of the highly optimized ESSL [17].

Beowulf Cluster. The PC cluster used was a computer system consisting of one front-end PC and five dual processor PCs equipped with 350 MHz Pentium II processors. Communication was provided via fast Ethernet (100 Mbit/s). Theoretical peak performance using p processors was $p \times 350$ Mflop/s. HPF support was provided via Portland Group's *pghpf* compiler version 3.0-3. The local BLAS routines used were those developed for the ASCI Red project (see `http://www.cs.utk.edu/~ghenry/distrib/`).

4.2 Results

IBM SP2. Fig. 6 illustrates the floating-point performance on 8 processors of the IBM SP2 of (i) the newly developed Cholesky factorization routine `par_dpotrf`, and (ii) `SLHPF/la_posv` called from an HPF program using the SLHPF interface [8] for a two-dimensional data distribution CYCLIC(100) in both dimensions.

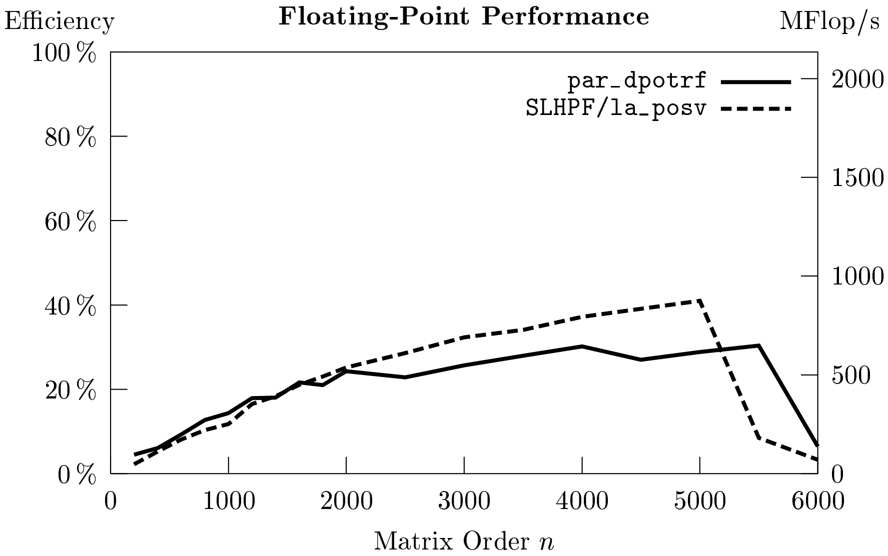

Figure 6: Cholesky factorization on 8 processors of the IBM SP2. The efficiency is given as the percentage of the peak performance $8 \times 267 = 2136$ Mflop/s.

For this problem, which requires a lot of communication, the HPF code combined with the BLAS is in the range of the corresponding SCALAPACK routine `pdpotrf` (see also Table 1). This shows that it is possible to achieve performance comparable with highly optimized message passing codes using HPF.

Table 2: Two different 2D FFT codes on $p = 1, 2, 4, 8, 16$ processors of the IBM SP2. The efficiency is given as the percentage of the peak performance $p \times 267$ Mflop/s.

Number of processors p	1	2	4	8	16
`par_fft`	15.7 %	12.1 %	11.2 %	10.3 %	9.2 %
Native HPF	6.5 %	5.4 %	5.3 %	5.1 %	3.4 %

Table 2 shows a comparison of a native HPF implementation of a 2D FFT algorithm with the HPF code `par_fft` based on a 1D FFTPACK routine using a matrix size of 2048. FFTPACK is one of the most efficient FFT packages (see Auer et al. [3]). In the native HPF version a Cooley-Tukey radix-2 FFT algorithm (Van Loan [30]) is used for the one-dimensional FFTs. The results show that the new method utilizing existing FFT software yields performance gains by a factor of two.

Beowulf Cluster. Fig. 7 compares the routines `par_dgemm` and `PBLAS/pdgemm` (called directly as in SCALAPACK) on the Beowulf cluster. In both cases a CYCLIC(100) data distribution was used in both dimensions. This distribution results in best performance with `PBLAS/pdgemm` and is also a favorable distribution for `par_dgemm`. Fig. 7 shows that HPF programs can be as efficient as subroutines from highly optimized parallel numerical libraries.

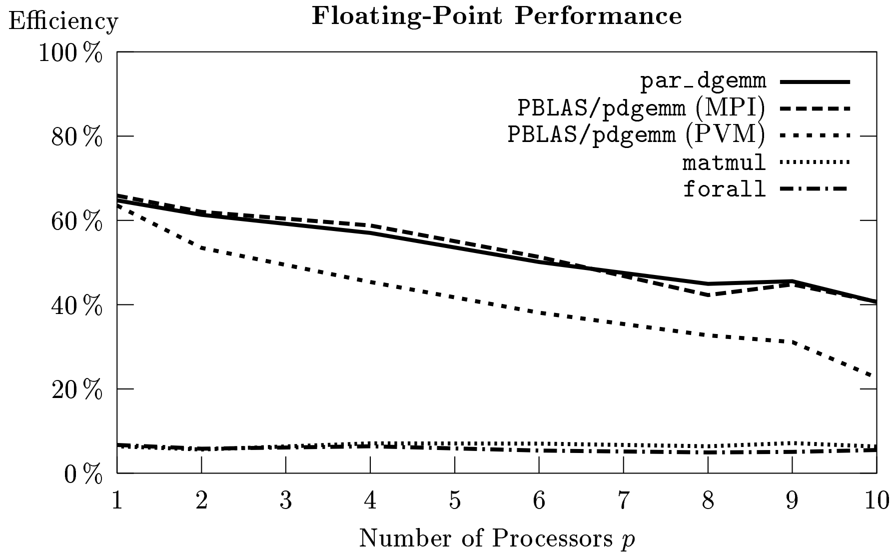

Figure 7: Multiplication of 3000×3000 matrices on p processors of a Beowulf cluster. The efficiency is given as the percentage of the peak performance $p \times 350$ Mflop/s.

Fig. 8 compares the routines `par_dsymm` and `PBLAS/pdsymm`. A CYCLIC(32) data distribution in both dimensions was used for both routines. 32 is the default block size chosen in SLHPF (see Blackford et al. [8]). The freedom to choose different parameters for algorithmic and distribution block size in `par_dsymm` (Version 1) obviously leads to higher performance than using just one fixed parameter for both purposes (Version 2).

Conclusion

In this paper, HPF is shown to support the construction of codes which are portable across important parallel architectures, while achieving highly satisfactory performance.

It has been demonstrated that subroutines of existing numerical high performance libraries can be integrated into HPF code in a portable and efficient way using HPF language features only. This new approach guarantees high local performance in HPF programs and yields significant performance improvements compared to native HPF implementations which do not take advantage of existing software packages and libraries.

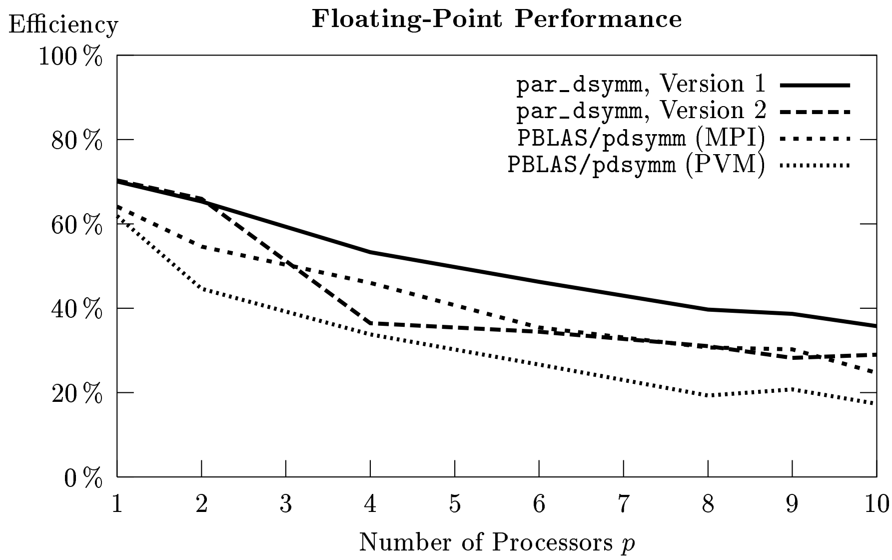

Figure 8: Symmetric matrix-matrix multiplication routines on a Beowulf cluster. The efficiency is given as the percentage of the peak performance $p \times 350$ Mflop/s.

Acknowledgements

We want to express our gratitude to John Merlin for many helpful discussions, to Herbert Karner for providing us with knowledge and example codes for Section 3.4 and Table 2, and to the Austrian Science Fund (FWF) for financial support.

References

[1] B. S. Andersen, F. Gustavson, J. Wasniewski, P. Y. Yalamov: *Recursive Formulation of Some Dense Linear Algebra Algorithms.* In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM Press, Philadelphia, PA, 1999.

[2] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. C. Sorensen: LAPACK *Users' Guide*, 3rd edn. SIAM Press, Philadelphia, PA, 1999.

[3] M. Auer, R. Benedik, F. Franchetti, H. Karner, P. Kristöfel, R. Schachinger, A. Slateff, C. W. Ueberhuber: *Performance Evaluation of FFT Routines—Machine Independent Serial Programs.* Technical Report AURORA TR1999-05, Vienna University of Technology, 1999. `www.vcpc.univie.ac.at/aurora/publications/`

[4] S. Benkner: *HPF+: High Performance Fortran for Advanced Scientific and Engineering Applications.* Future Generation Computer Systems 15 (1999), pp. 381–391.

[5] S. Benkner: *Optimizing Irregular HPF Applications Using Halos.* Concurrency: Practice and Experience 12 (2000).

[6] J. Bilmes, K. Asanovic, C. W. Chin, J. W. Demmel: *Optimizing Matrix Multiply using* PHIPAC*: a Portable, High-Performance, ANSI C Coding Methodology.* In *Proceedings of the International Conference on Supercomputing*, ACM, Vienna, Austria, 1997, pp. 340–347. Also LAPACK Working Note 111.

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

[8] L. S. Blackford, J. J. Dongarra, C. A. Papadopoulos, R. C. Whaley: *Installation Guide and Design of the HPF 1.1 Interface to* ScaLapack, *SLHPF*. Lapack Working Note 137, University of Tennessee, Knoxville, TN, 1998.

[9] T. Brandes, D. Greco: *Realization of an HPF Interface to* ScaLapack *with Redistributions*. In *High-Performance Computing and Networking. International Conference and Exhibition*, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1996, pp. 834–839.

[10] R. Chandra, D. Kohr, R. Menon, L. Dagum, D. Maydan, J. McDonald: *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, 2000.

[11] B. M. Chapman, H. P. Zima, M. Haines, P. Mehrotra, J. Van Rosendale: *OPUS: A Coordination Language for Multidisciplinary Applications*. Scientific Programming 6/4 (1997), pp. 345–362.

[12] R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, A. Sussman: *Applying the CHAOS/PARTI Library to Irregular Problems in Computation Chemistry and Computational Aerodynamics*. Proceedings of the 1993 Scalable Parallell Libraries Conference, IEEE Computer Society Press (1993), pp. 45–56.

[13] J. J. Dongarra, J. Du Croz, I. S. Duff, S. Hammarling: *A Set of Level 3 Basic Linear Algebra Subprograms*. ACM Trans. Math. Softw. 16 (1990), pp. 1–17, 18–28.

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson: *An Extended Set of Basic Linear Algebra Subprograms*. ACM Trans. Math. Software 14 (1988), pp. 18–32.

[15] H. J. Ehold, W. N. Gansterer, D. F. Kvasnicka, C. W. Ueberhuber: *HPF and Numerical Libraries*. In *Parallel Computation. Proceedings of the 4th International ACPC Conference* (P. Zinterhof, M. Vajteršic, A. Uhl, eds.), Springer-Verlag, Berlin Heidelberg New York Tokyo, 1999, Vol. 1557 of *Lecture Notes in Computer Science*, pp. 140–152.

[16] H. J. Ehold, W. N. Gansterer, C. W. Ueberhuber: *HPF—State of the Art*. Technical Report AURORA TR1998-01, Vienna University of Technology, 1998.
`www.vcpc.univie.ac.at/aurora/publications/`

[17] ESSL: *Engineering and Scientific Subroutine Library for AIX*. IBM, 1998.
`www.rs6000.ibm.com/resource/aix_resource/sp_books/essl/`

[18] HPF-Forum: *High Performance Fortran Language Specification Version 1.0*. Scientific Programming 2 (1) (1993), pp. 1–170.
`www.crpc.rice.edu/HPFF/hpf1/`

[19] HPF-Forum: *High Performance Fortran Language Specification Version 2.0*, 1997.
`www.crpc.rice.edu/HPFF/hpf2/`

[20] C. L. Lawson, R. J. Hanson, F. T. Krogh, D. R. Kincaid: *Basic Linear Algebra Subprograms for Fortran Usage*. ACM Trans. Math. Software 5 (1979), pp. 308–323.

[21] P. A. R. Lorenzo, A. Müller, Y. Murakami, B. J. N. Wylie: *HPF Interface to* ScaLapack. In *Third International Workshop PARA '96*, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1996, pp. 457–466.

[22] P. Mehrotra, J. Van Rosendale, H. P. Zima: *Language Support for Multidisciplinary Applications.* IEEE Computation Science and Engineering 5/2 (1998), pp. 64–75.

[23] MPI-Forum: *MPI: A Message Passing Interface Standard*, 1994.
`www.epm.ornl.gov/~walker/mpi/index.html`

[24] M. Noelle, N. Jungclaus: *Efficient Implementation of FFT-Like Algorithms on MIMD Systems.* In *Proceedings of SUSIPCO-94, the 7th European Signal Processing Conference* (M. J. J. Holt et al. , ed.), EURASIP, Lausanne, 1994, Vol. III, pp. 1625–1628.

[25] OpenMP Forum: *OpenMP Fortran Application Program Interface.* Version 1.1, 1999, `http://www.openmp.org`.

[26] J. M. Ortega: *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, New York NY, 1988.

[27] J. Saltz, R. Ponnusamy, S. D. Sharma, B. Moon, Y.-S. Hwang, M. Uysal, R. Das: *A Manual for the CHAOS Runtime Library.* Technical Report, University of Maryland, Institute for Advanced Computer Studies, Dept. of Computer Science, CS-TR-3437, 1995.

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

[29] R. van de Geijn: *Using* PLAPACK*: Parallel Linear Algebra Package*. The MIT Press, Cambridge, MA, 1997.

[30] C. F. Van Loan: *Computational Frameworks for the Fast Fourier Transform.* SIAM Press, Philadelphia, PA, 1992.

[31] R. C. Whaley, J.J. Dongarra: *Automatically Tuned Linear Algebra Software.* LAPACK Working Note 131, University of Tennessee, Knoxville, TN, 1997.

[32] R. C. Whaley, A. Petitet, J. J. Dongarra: *Automated Empirical Optimization of Software and the ATLAS Project.* LAPACK Working Note 147, University of Tennessee, Knoxville, TN, 2000.