High Local Performance in HPF Codes

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

[1] Institute for Applied and Numerical Mathematics,
Vienna University of Technology, Austria.
{ehold,ganst,christof}@aurora.tuwien.ac.at

[2] Institute for Physical and Theoretical Chemistry,
Vienna University of Technology, Austria.
dieter.kvasnicka@tuwien.ac.at

March 2000

AURORA TR2000-06

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

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 in a highly efficient manner. The method suggested utilizes HPF's EXTRINSIC mechanism to integrate existing numerical *single processor software* for computationally expensive kernels into HPF programs. By using our technique, the empirical efficiency, i. e., the ratio of the empirical floating-point performance to the theoretical peak performance, can be driven up to 50 % and more. Even on message passing machines with slow communication networks, such as PC clusters (Beowulf clusters) using 100 Mbit/s Ethernet interconnection, empirical efficiency results at a highly satisfactory level can be obtained. 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.

Contents

Chapter 1

Introduction

High Performance Fortran (HPF [16]) 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 [15] in 1992 by the High Performance Fortran Forum two important trends 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. This was mainly due to the fact that for quite a long time HPF's success was unclear. Not even the success of Fortran 90/95, being the basis of HPF, could be taken for granted.

- Users, on the other hand, soon started to demand more and advanced features to deal with a broader range of applications than just regular data-parallel ones. In particular, HPF support for irregular distributions and for task parallelism was demanded (see, for instance, 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 [16].

HPF is a conceptually very elegant and attractive approach to 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 easier and requires less coding effort than explicitly programming message-passing, for example, using MPI [19]. Nevertheless, so far HPF has not become widely accepted by users for several reasons (Ehold et al. [13]).

- It took a long time until the language standard HPF 2.0 [16] was fully supported by commercial compilers. Only recently mature compilers have become available. Unfortunately, several companies have suspended their activities of HPF compiler development because the market for HPF compilers has been assessed 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, especially for *local* computations.

Related Work

Related previous work has concentrated on interfacing HPF to extrinsic *parallel* library routines from numerical software packages such as ScaLapack [7]. In contrast to the main focus of this paper, in this case the extrinsic library routine itself is responsible for communication as well as computation. 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 extrinsic library routines.

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 Blacs and ScaLapack calls. In contrast to this approach, which requires 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. The current 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` (Blas [10]), 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. [18] 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 allows for bypassing the EXTRINSIC mechanism of HPF.

Synopsis

In this paper it is shown that in many cases HPF programs utilizing *sequential* library routines are preferable to HPF programs utilizing *parallel* libraries. This is not only due to the fact that most of the HPF constructs required in the former case are already part of standard HPF and therefore highly portable. In terms of performance, the first approach also can be made to be less sensitive to the data distribution.

In Chapter 2 the basic methodology for integrating sequential library routines into HPF programs is introduced. In Chapter 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-matrix multiplication, Cholesky factorization, and two-dimensional FFT. In Chapter 4 numerical experiments carried out on various parallel computers are reported.

Chapter 2

HPF and Numerical Libraries

A considerable amount of expertise has been incorporated into high quality software like the BLAS [10, 11, 17] and into packages which use the BLAS as building blocks like LAPACK [2], the standard sequential package for dense or banded matrices, and parallel packages such as SCALAPACK [7] or PLAPACK [23]. It is hardly possible to achieve comparable floating-point performance when coding in HPF directly.

Native[1] HPF code executing standard numerical operations often performs very poorly and only achieves a very low utilization of the potential floating-point power of a given computer system. Take, as an example, the floating-point performance of two different native HPF implementations of matrix-matrix multiplication (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. 2.1. The performance of the sequential BLAS routine is equal or better than that of native HPF codes running on 10 processors.

As another example, various implementations of Cholesky factorization have been compared (see Table 2.1). Here again it turned out that the sequential LAPACK routine `LAPACK/dpotrf` is more than 10 times faster than a native HPF implementation whose parallel speed-up is nearly linear. The sequential LAPACK routine is still 1.5 times faster than the parallel HPF code running on 8 processors! However, a newly developed parallel HPF routine utilizing sequentially optimized BLAS routines (denoted "HPF + BLAS", see Section 3.3) is even faster than the corresponding SCALAPACK routine and shows a similar speed-up as this highly tuned routine.

Utilization of available high quality software is desirable not only in terms of performance but also from a very practical point of view: in many cases a

[1] HPF routines which do not use the EXTRINSIC mechanism will be called *native*.

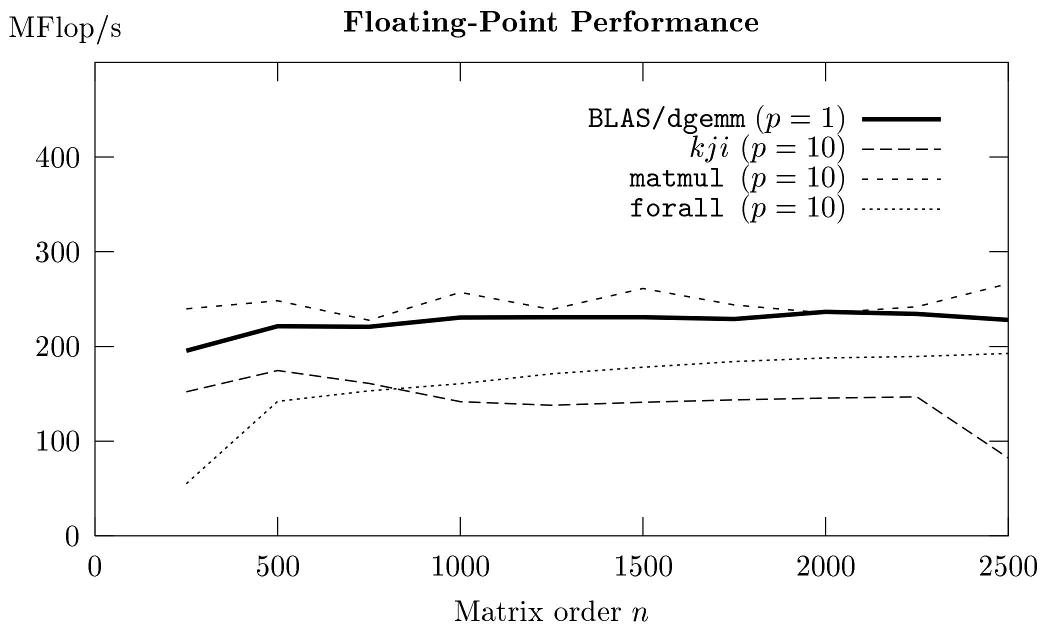

Figure 2.1: Matrix-matrix multiplication on a Beowulf Cluster whose peak performance is $p \times 350$ Mflop/s.

Table 2.1: Cholesky factorization of a symmetric 3000×3000 matrix on a Beowulf cluster (see Section 4.1) using (i) an LAPACK routine, (ii) a native HPF routine, (iii) a SCALAPACK routine, and (iv) a 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
HPF + BLAS	48.6 s	34.2 s	25.3 s	18.4 s

considerable amount of programming and debugging effort can be saved by using existing library routines as building blocks instead of coding algorithms in HPF directly, in particular for applications involving compute intensive mathematical operations. Thus, when HPF is used for parallelization, the necessity of combining routines from numerical packages and libraries with HPF code arises quite naturally.

2.1 Optimizing Local Performance

Even if a problem is well suited for an HPF implementation, performance of the code heavily relies on the maturity of HPF compilers, which cannot be taken for granted at present (Ehold et al. [13]). Even if an HPF compiler does a good job organizing data distribution and communication between processors, the actual performance achieved can be disappointingly low. According to our experience, one of the key issues to improve overall parallel performance in HPF is to optimize 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 accesible 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 [25]). These code generators automatically find the best choice of hardware dependent parameters and produce highly efficient implementations of the Level 3 Blas.

Clearly, HPF users should be able to take advantage of such resources. Highly optimized, sequential software should be utilized in order 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 remains responsible for handling and organizing parallelism supported by user directives.

The basic concept and main ideas presented in this paper are applicable to many data-parallel mathematical operations (mostly, but not exclusively, from linear algebra).

2.2 HPF Features Needed for High Local Performance

The EXTRINSIC mechanism (HPF 2.0 [16]) is the basic facility for integrating procedures from other programming languages or models into HPF code.

The concept for achieving high local performance presented in this paper is based on this mechanism. It is *portable* as it only relies on standard HPF features and, additionally, on very few specific features of the HPF 2.0 Approved Extensions (HPF 2.0 [16]). The required HPF features are

- EXTRINSIC(F77_LOCAL) subroutines;
- EXTRINSIC(HPF_LOCAL) subroutines;
- the INHERIT directive or TEMPLATES;
- an advanced form of the ALIGN directive, namely `ALIGN A(j,*) WITH B(j,*)` (replication of `A` along one dimension of `B`) used for linear algebra algorithms.

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

- The index conversion subroutine LOCAL_TO_GLOBAL, which is included in the HPF_LOCAL_LIBRARY, is used for accessing symmetric matrices;
- The inquiry subroutine HPF_DISTRIBUTION is used for obtaining mapping information of the arrays involved.

F77_LOCAL. The extrinsic kind F77_LOCAL refers to procedures that are implemented in Fortran 77 and in the *local* programming model. Each processor executes this 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 is required for determining the size of *local* arrays, is not supported within a F77_LOCAL subroutine, HPF_LOCAL has to be used. Since the EXTRINSIC kind F77_LOCAL is not part of the HPF 2.0 standard, but is only included in the Approved Extensions, its actual implementation varies to some extent across the available HPF compilers.

HPF_LOCAL. The extrinsic kind HPF_LOCAL refers to procedures implemented in HPF in the local programming model. This means that each processor executes the HPF code only on the local portions of distributed data. Specifically, it provides a means of handing arguments to local BLAS routines.

INHERIT. The INHERIT directive tells the compiler that the distribution of each element of an array should remain identical when this array or a subsection of it is passed to a subroutine.

ALIGN. Proper alignment of submatrices is one of the most important features of the new method. Small work arrays are aligned properly (along one dimension) with a given distributed array, so that a subsequent copy of rectangular blocks makes sure that all data is resident on all processors where it is required. Data is usually required on more than one processor at once. Therefore, the work arrays are aligned only in one dimension and replicated in the other dimension (see Fig. 3.1). This can be specified with an advanced form of the ALIGN statement and makes HPF programs which utilize optimized extrinsic routines short and elegant, while working without any modification for many different data distributions.

LOCAL_TO_GLOBAL. In some routines it is necessary to treat certain matrix elements or subblocks (for example, diagonal elements) differently than others. When a distributed array is passed to an HPF_LOCAL or a F77_LOCAL routine then each processor sees only the local parts of this array. Each processor references the array elements using *local* indices. The routine LOCAL_TO_GLOBAL allows to inquire locally the global indices of these array elements. In Section 3.2 this will be required to determine whether these array elements are above, at, or below the matrix diagonal.

HPF_DISTRIBUTION. Some of the routines implemented require a certain data distribution (for example, `par_dpotrf` requires a two-dimensional block-cyclic data layout) for optimum performance. The inquiry subroutine HPF_DISTRIBUTION is used to determine whether the data is distributed suitably and to inquire the block size of the current distribution, which is then also used as the algorithmic block size.

Chapter 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[1] routine for computing 1D FFTs locally.

3.1 General Matrix-Matrix Multiplication

An HPF routine called `par_dgemm` (*parallel general matrix-matrix multiplication*) has been developed, which computes the product $C \in \mathbb{R}^{m \times n}$ of two general matrices $A \in \mathbb{R}^{m \times l}$, $B \in \mathbb{R}^{l \times n}$. 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, choosing the proper loop order and replicating some submatrices allows to localize *all* of the computation involved.

3.1.1 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

[1] `http://www.netlib.no/netlib/fftpack/`

variants (see, for instance, Ueberhuber [22]) having different parallelization characteristics and consequently requiring different means for expressing parallelism (see, for instance, Ortega [21]). 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 the necessity for communicating the elements of C. Moreover, varying k the outer loop is well suited for two-dimensional data distribution usually preferred for matrix operations (see, for example, ScaLapack [7]).

3.1.2 PAR_DGEMM

For performance reasons, a *blocked* version of matrix-matrix multiplication (Ueberhuber [22]) 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 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. 3.1), 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 result, the routine becomes independent of the prior distributions of A and B and all communication takes place in the outermost loop.

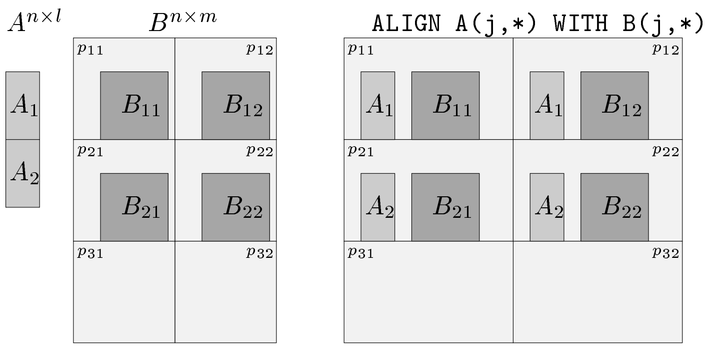

Figure 3.1: Partial replication of the array A adapted to the distribution of the array B over the processors p_{ij}.

Inside the k-loop the HPF_LOCAL routine `local_dgemm` is called. In addition to the blocks of A and B `local_dgemm` takes also their size (i. e., the block size b of the algorithm) as an argument and computes the outer products 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 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`: $C = C + AA^T$).

3.2.1 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 concerns the handling of the symmetric matrix A. In `par_dgemm` the alignment of the elements of A to C is achieved by copying an array section. In `par_dsymm` where the symmetry of A has to be taken into account, the copy operation is split into three parts (see Fig. 3.2).

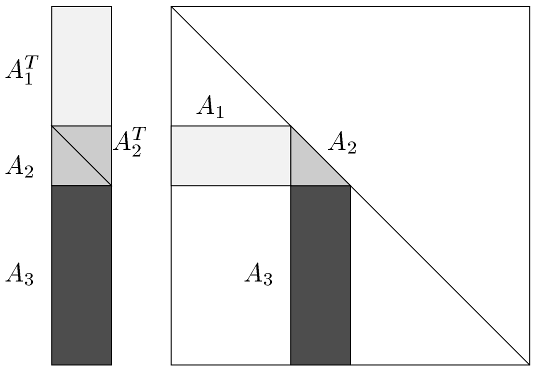

Figure 3.2: Threefold copying in `par_dsymm`.

As in the general case, 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 equivalent to those of `par_dgemm`.

Beside the simplicity of developing a new parallel routine out of an existing one, the implementation presented here also allows to tune the distribution block size independently of the algorithmic block size. This gives the programmer more flexibility in performance tuning than the ScaLapack routines do.

3.2.2 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, $C := \alpha AA^T + \beta C$ 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 concerning its local elements. Special care has to be taken with respect to load distribution. In contrast to `par_dgemm` and `par_dsymm` two-dimensional block-cyclic data distribution is clearly the best choice which leads to a reasonable load distribution.

New problems arise concerning the shape of local portions of C (see Fig. 3.3). In contrast to `par_dgemm` and `par_dsymm` the operations on the local portions cannot be combined into one single Blas call. The solution is to combine only subblocks of each column block in a call to `BLAS/dgemm` and to compute diagonal blocks by 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. The distribution of A is done, similar to `par_dgemm` and `par_dsymm`, outside the HPF_LOCAL routine. If A is too large, an additional (higher level) loop is used to split up A into column blocks with are treated sequentially.

To obtain a slightly better performance, operations on several local column blocks, at least their major parts, might be combined in one `BLAS/dgemm` call. However, for the sake of elegance and simplicity, this variant was not 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

Computing the Cholesky factorization $LL^\top$ of a real symmetric positive definite matrix A serves as another example. Early experiments with native HPF implementations of the Cholesky factorization on a Meiko CS-2 have

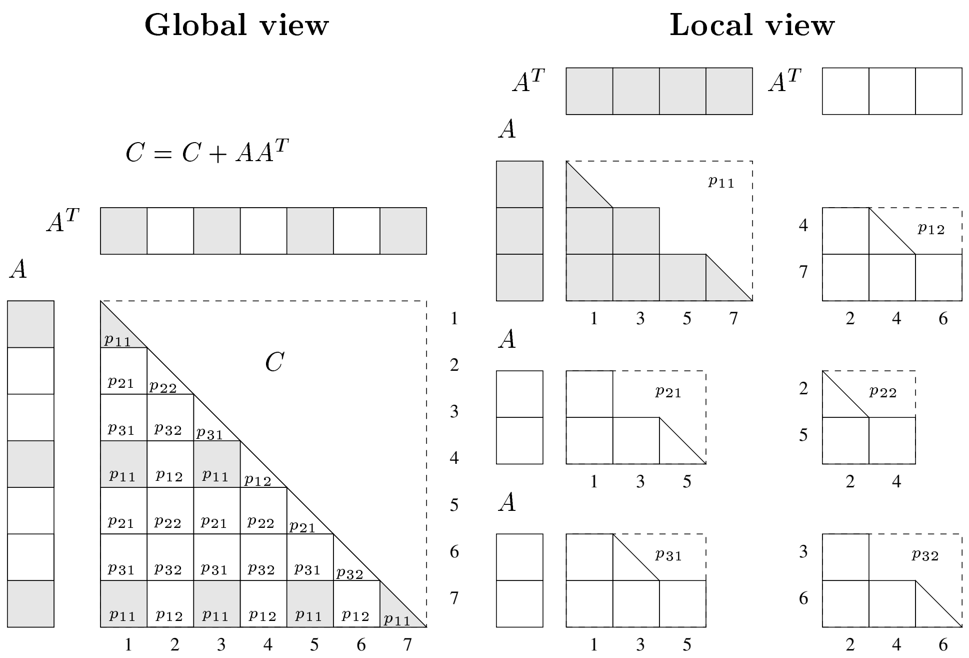

Figure 3.3: Two-dimensional block-cyclic distribution of matrix C on a 3×2 processor grid p_{ij}. Alignment of matrix A with matrix C.

shown very poor performance. In fact, the best performance using an HPF code was achieved on 16 processors, but it was still slower than the routine `LAPACK/dpotrf` on one processor of the same machine (cf. Table 2.1 and Ehold et al. [12]).

3.3.1 Parallelization

Blocked Cholesky factorization exhibits better locality of reference and is therefore preferred over the unblocked version on modern computer architectures (Anderson et al. [2], Ueberhuber [22]). 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. 3.4).

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 (Step 1 is applied recursively), (ii) right-looking (Step 3 is applied recursively), and (iii) "recursive" Cholesky factorization (Steps 1 *and* 3 are applied recursively; see Andersen et al. [1]). The right-looking algorithm is used in this paper.

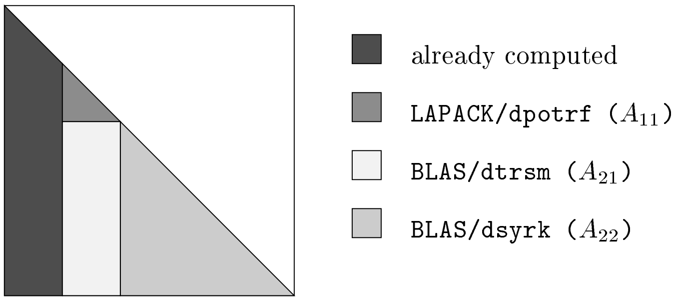

Figure 3.4: Update areas of blocked Cholesky factorization.

3.3.2 Implementation

In the HPF+BLAS version Step 1 (using `LAPACK/dpotrf`) —which is computationally negligible—is done sequentially on the processor which owns the block A_{11}, using the HPF extrinsic kind F77_LOCAL. `BLAS/dtrsm` is called on the processors that own local data of the block A_{21}. Since block A_{11} is needed by all participating processors this block is replicated on all those processors (like the matrices A and B in `par_dgemm`, Section 3.1). The matrix update `BLAS/dsyrk` (Step 3) is called via the HPF wrapper routine `par_dsyrk` (see Section 3.2.2). The Cholesky factorization of Step 3 is applied recursively.

If the submatrix A_{11} is of order b, the appropriate distribution of the matrix A is (CYCLIC(kb), CYCLIC(lb)) with $k, l \in \mathbb{N}$. This distribution makes sure that submatrix A_{11} is local on one processor and that A_{21} is owned by one processor column only. For simplicity it is assumed that the block size b divides n, the dimension of matrix A, evenly.

SCALAPACK, native HPF, and the HPF+BLAS routine use the right-looking algorithm. It should be noted that (i) the HPF+BLAS code 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 computed by calculating one-dimensional FFTs for each column and then for each row. Consequently, there is 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 be able to also compute the rowwise one-dimensional FFTs locally. The transposition involves all-to-all communication leading to considerable communication overhead. Nevertheless, this approach is more efficient than an algorithm based on a static decomposition and parallel one-dimensional FFTs to perform the FFTs along the rows (see, for example, Noelle and Jungclaus [20]).

Chapter 4

Numerical Experiments

In this chapter some considerations concerning data layout, a short overview of the used hardware platforms and a description of the results of the performed experiments are given.

4.1 Platforms

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

4.1.1 IBM SP2

The SP2 used in the experiments is a system equipped with 37 processors (POWER2 architecture RS/6000), of which up to 16 at a time were used in the experiments. Communication is provided via the SP2's high performance switch. Theoretical peak performance using p processors is $p \times 267$ Mflop/s. HPF support is provided via Portland Group's HPF compiler *pghpf* version 2.4-4. The local BLAS routines used were part of the highly optimized ESSL [14].

4.1.2 Beowulf Cluster

The PC cluster used is a computer system consisting of one front-end PC and five dual processor PCs equipped with 350 MHz Pentium II processors. Communication is provided via a Fast Ethernet network (100 Mbit/s). Theoretical peak performance using p processors is $p \times 350$ Mflop/s. HPF support is provided via Portland Group's *pghpf* compiler version 3.0-3. The local BLAS routines used are those developed for the ASCI Red project.[1]

4.2 Results

4.2.1 IBM SP2

Fig. 4.1 illustrates the floating-point performance on 8 processors of an 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.

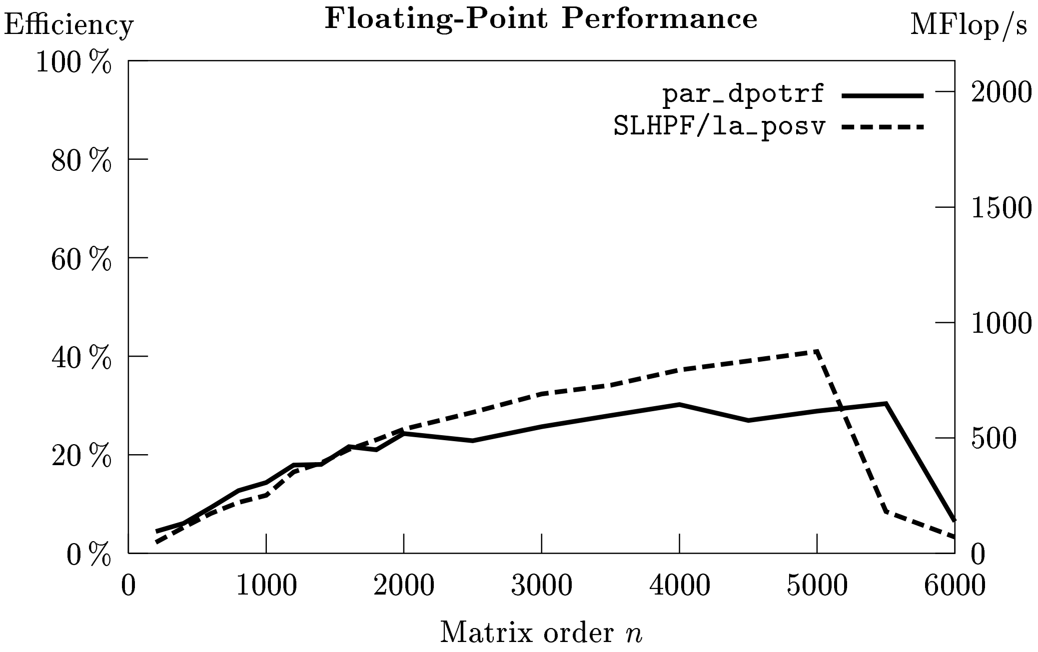

Figure 4.1: Cholesky factorization on 8 processors of an IBM SP2. Efficiency is given as the percentage of the peak performance $8 \times 267 = 2136$ Mflop/s.

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 (see Table 2.1). This shows that it is possible to achieve performance that is comparable with highly optimized message passing codes using HPF.

[1] `http://www.cs.utk.edu/~ghenry/distrib/`

Table 4.1: Efficiency of two different 2D FFT codes running on p processors of an IBM SP2. 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 4.1 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 [24]) 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.

4.2.2 Beowulf Cluster

Fig. 4.2 compares the HPF+BLAS routine `par_dgemm` with the SCALAPACK routine `PBLAS/pdgemm` (called directly as in SCALAPACK). 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 very favorable distribution for `par_dgemm`. Fig. 4.2 shows that HPF programs can be as efficient as subroutines from highly optimized parallel numerical libraries.

Fig. 4.3 compares the HPF+BLAS routine `par_dsymm` with the SCALAPACK routine `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 of choosing different parameters for algorithmic and distribution block size (Version 1) obviously leads to higher performance than choosing one fixed parameter for both purposes (Version 2).

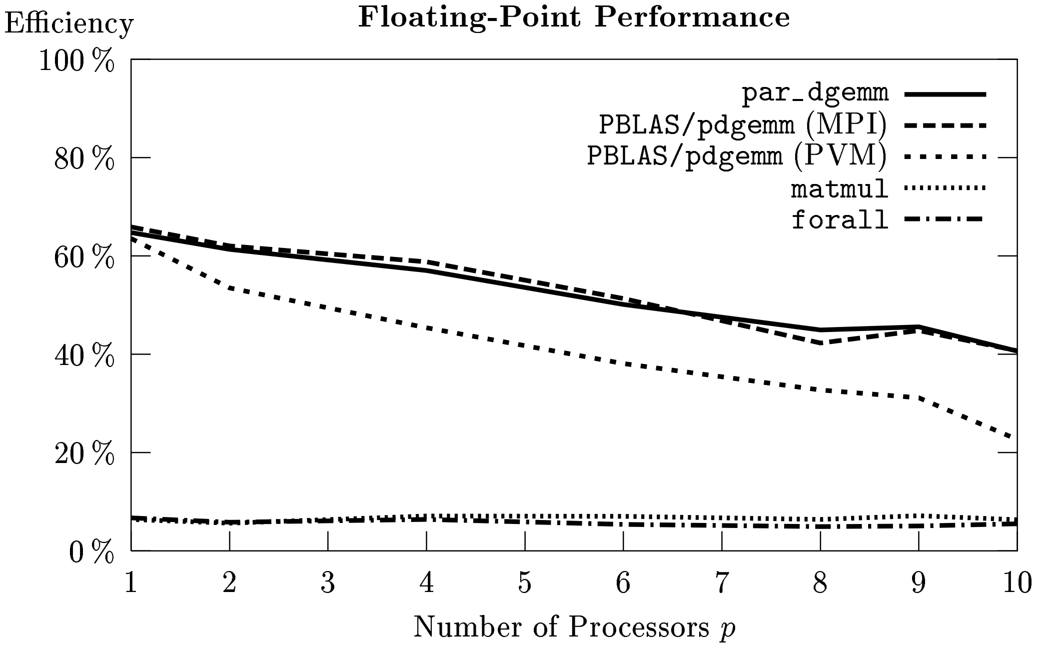

Figure 4.2: Matrix-matrix multiplication on a Beowulf cluster. Efficiency is given as the percentage of the peak performance $p \times 350$ Mflop/s.

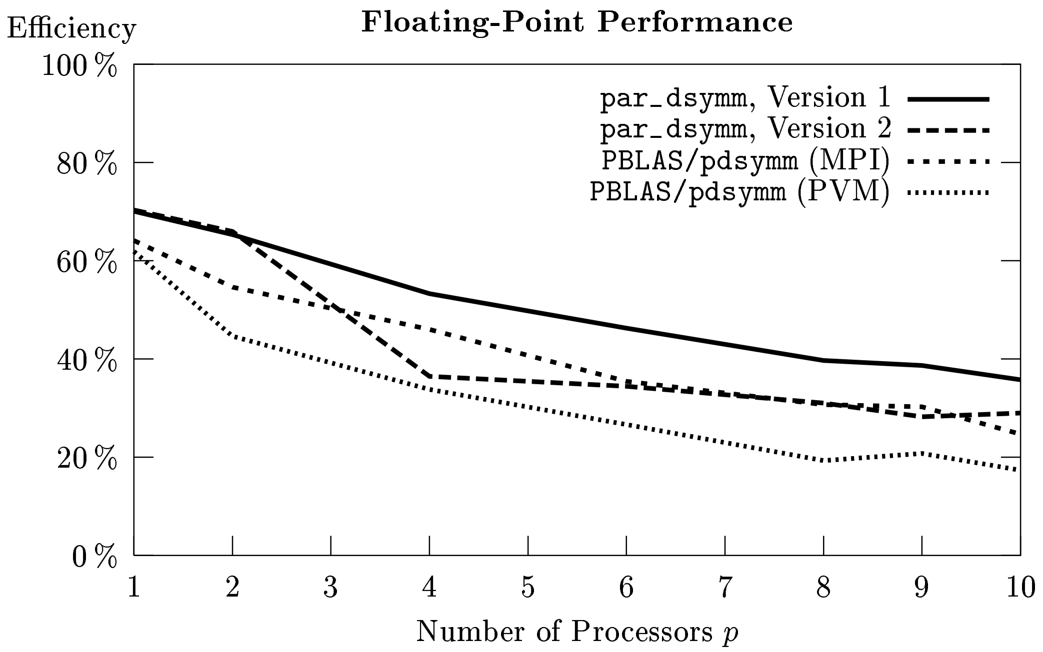

Figure 4.3: Symmetric matrix-matrix multiplication on a Beowulf cluster. Efficiency is given as the percentage of the peak performance $p \times 350$ Mflop/s.

Conclusion

In this paper, HPF is shown to support construction of codes portable across important parallel architectures, while achieving a highly satisfactory floating-point 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. The 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.

The price/performance ratio of 11 US$ per Mflop/s, achieved on a small PC cluster, is better than the price/performance achievement which was awarded the Gordon Bell Prize in 1998.

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 4.1, and to the Austrian Science Fund (FWF) for financial support.

Bibliography

[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] 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.

[11] 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.

[12] 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.

[13] 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/`

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

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

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

[17] 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.

[18] 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.

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

[20] 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., eds.), EURASIP, Lausanne, 1994, vol. III, pp. 1625–1628.

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

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

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

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

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