First experiences with high performance Fortran on the Intel Paragon

Report

Author(s):
Sturler, Eric de; Strumpen, Volker

Publication date:
1995

Permanent link:
https://doi.org/10.3929/ethz-a-006651254

Originally published in:
Internal report 234

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Wissenschaftliches Rechnen

Eric De Sturler
Volker Strumpen

First Experiences with High Performance Fortran on the Intel Paragon

May 1995

234

ETH Zürich
Departement Informatik
Institut für Wissenschaftliches Rechnen
Prof. Dr. W. Gander

Eric De Sturler
Interdisziplinäres Projektzentrum für Supercomputing
Eidgenössische Technische Hochschule
CH-8092 Zürich
e-mail: sturler@ips.id.ethz.ch

Volker Strumpen
Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich
e-mail: strumpen@inf.ethz.ch

This report is also available via anonymous ftp from ftp.inf.ethz.ch
as doc/tech-reports/1995/234.ps.

First Experiences with High Performance Fortran on the Intel Paragon

Eric De Sturler
sturler@ips.id.ethz.ch
Interdisciplinary Project Center for Supercomputing

Volker Strumpen
strumpen@inf.ethz.ch
Institute for Scientific Computing

Swiss Federal Institute of Technology
ETH-Zentrum, CH-8092 Zurich, Switzerland

May 9, 1995

Abstract

Recently the first commercial HPF subset compilers have appeared. This paper reports on our experiences with the APR XHPF compiler, version 1.2, for the Intel Paragon. At this stage, we do not expect optimum performance of our HPF programs, even though performance will eventually be of paramount importance for the acceptance of HPF. Instead, the objective is to study how to convert large F77 programs to HPF such that the compiler generates efficient parallel code.

We use three case studies to identify several problems; we discuss our solutions at the program level, and present the results on the Intel Paragon. We use the dense matrix-matrix product to show that the distribution of arrays and the order of nested loops significantly influence the performance of the parallel program. This influence is of great importance, because loop parallelization and array distribution are the corner stones of HPF. We use Gaussian elimination to study the parallelization strategy of the compiler if the (optimal) parallelization is not clear from the data distribution. This provides some insights into the analysis and optimization capabilities of the compiler, and shows how much effort is required from the programmer to get an efficient parallel implementation. Finally, we use a small application to show that the interprocedural analysis introduces problems for the parallelization, even though all subroutines of the application are easy to parallelize by themselves. The application consists of a finite volume discretization on a structured grid and a nested iterative solver. The problems arise when arrays with alignment attributes or with distribution attributes are passed to several subroutines, in which alignment and distribution of the dummy arguments are declared, and when in addition such subroutines are called at several places in the program with different arguments.

1 Introduction

Our objective is the conversion of large F77 programs to HPF programs that allow the compiler to generate efficient parallel code. However, our conclusions apply to the development of new HPF programs as well. We do not know in detail how APR's FORGE HPF Parallelizer XHPF [5] works below the user level, nor whether other compilers will handle certain problems the same way. However, we focus on problems and solutions that seem to be at the core of using

HPF, and we believe that the problems and solutions that we have found are valuable for other programmers and also for compiler developers.

HPF is a data-parallel language where parallelization is addressed by array distribution and loop parallelization. The programmer specifies the distribution of arrays and alignment between arrays. Furthermore, HPF offers the `forall` statement and several new intrinsic functions. Except for these explicit parallel constructs, the compiler is expected to generate reasonably efficient parallel code automatically. An important aspect of HPF directives is that they are *safe*; they do not change the semantics of a program.

Apart from the HPF subset, the APR XHPF compiler offers several additional directives. The most important use of the APR directives is to force the compiler to make certain optimizations, like forced loop parallelization, or neglecting communication or synchronization. In contrast to HPF directives, these directives are *dangerous*; they may change the semantics of the program, which makes their use undesirable in general. However, it seems that with the state-of-the-art in parallel compiler technology these directives are necessary to generate efficient parallel code, and as such they are a valuable aid if used sparingly. We refer to these directives collectively as *forced optimizations*.

2 Case studies

We analyze HPF using three case studies. (1) With a dense matrix-matrix product we show that the distribution of arrays and the order of nested loops in which distributed arrays are referenced significantly influence the compiler optimizations and the performance of the parallel program. More specifically, they affect the place of subroutine calls for communication and runtime checks in the program generated by the HPF compiler, which determines the cost of communication. Furthermore, the subroutine calls reduce the optimization opportunities for the native compiler. (2) With Gaussian elimination with partial pivoting and backward substitution we study the parallelization strategies of the compiler, and we describe data-parallel implementations that help the compiler generate an efficient parallel implementation. (3) With a small application we show how unwanted effects of the interprocedural analysis may actually impede efficient parallelization.

2.1 Dense Matrix-Matrix Multiplication

The dense matrix-matrix product is one of the basic building blocks of linear algebra. To ensure scalability with respect to memory usage we distribute all three matrices. More specifically, we declare a row-block distribution for all matrices. In Figure 1 we give the ijk-variant of the threefold loop that implements the multiplication, including data distribution directives. All timings listed in this section give exclusively the runtime of the multiplication kernel (loop 40) or one of its permutations. These permutations are defined by exchanging lines 9–11.

We study two efficiency-related issues of the parallel program generated by HPF:

- **Loop partition match**: Consistency of the data distribution with the loop order and loop partition.

  Depending on the loop (nesting) order and the (given) data distribution the compiler decides where to insert runtime checks, index transformations, and communication primitives. This determines the number of messages generated, and hence affects the runtime efficiency. Given the data distribution, the appropriate order of the three nested loops in the sequential program is essential to ensure minimal communication overhead, and vice

versa. We call this aspect the *loop partition match* to characterize the match of the data distribution against the loop order and loop partition.

- **Optimization gap**: Loss of optimization potential of the parallel program generated by the HPF compiler.

  Efficiency aspects related to the processor architecture, in particular, register allocation, pipelining, and caching, are coupled with the loop order. Complex RISC processors such as the i860 of the Paragon rely heavily on machine specific optimizing compilers for generating efficient sequential code [2]. Thus, in our case the quality of the sequential code for the individual processors depends on the ability of the native compiler to optimize the program generated by the HPF compiler. We call the interface problems between the two compilers the *optimization gap*.

Although the optimization gap is not HPF specific, there are two reasons to mention this aspect explicitly. The first reason is the large difference in performance between the HPF programs and the sequential program on one processor, and the second reason is the difference in performance between the HPF programs with different loop orders on multiple processors. A further complication is that loop partition match and optimization gap are closely coupled, as will be illustrated by the performance results below.

```
      double precision a(m,n), b(n,m), c(m,m)
*
chpf$ distribute a (block,*)
chpf$ distribute b (block,*)
chpf$ distribute c (block,*)
*
      ... Initialize a, b and c
*
      do 40 i = 1, m
         do 40 j = 1, m
            do 40 k = 1, n
               c(i,j) = c(i,j) + a(i,k) * b(k,j)
   40 continue
```

Figure 1: Program fragment of matrix-matrix multiplication.

The optimization gap is illustrated in Table 1. Measurements with the six permutations of the loop order are shown; i traverses the rows of the product matrix C, j the columns, and k is the loop index of the inner product. The XHPF generated parallel program has been compiled with the native `if77` compiler using two different optimization levels. Option `-O` optimizes some register allocation and performs global optimizations such as induction recognition and loop invariant motion. Optimization level `-O4` introduces software pipelining, `-Mvect` enables vector optimizations, and `-Mstreamall` streams all vectors to and from cache in a vector loop. Such optimizations cannot be made manually in the F77 source program.

Comparing the run-times with little and full compiler optimization in Table 1, we observe two points. First, for the sequential code the loop order has a large influence on performance unless the native compiler (if77) can restructure the loop order. This is done when full optimization is

Loop order	-O		-full	
	seq	par-1	seq	par-1
ijk	159.91	160.22	10.45	138.14
ikj	272.31	272.57	10.45	276.88
jik	157.26	158.67	10.45	135.66
jki	42.19	43.44	10.45	31.96
kij	272.39	272.35	10.45	275.69
kji	44.50	45.08	10.45	35.12

Table 1: Runtimes [seconds] of sequential matrix-matrix multiplication (500×500): The sequential program (seq) and a 1-processor XHPF parallelized version (par-1) are compiled with different optimization degrees of Intel's native `if77` compiler. `-full` is an abbreviation of `-O4 -Mvect -Mstreamall`.

Loop order	-O	-full
$Cijk$	3.32	2.61
$Cikj$	6.65	6.75
$jCik$	9.03	8.14
$jRki$	21.02	20.78
$Rkij$	13.20	13.17
$Rkji$	8.20	7.90

Table 2: Runtimes [seconds] of different loop permutations of the parallel matrix-matrix multiplication (500×500) on 32 processors. All matrices are row-block distributed. `-full` is an abbreviation of `-O4 -Mvect -Mstreamall`.

enabled, which improves the computational speed by more than a factor of four, with the ikj and kij variants even up to a factor of 26. Second, for the parallel code executed on one processor we see that even with full optimization the native compiler is no longer able to restructure the loop order because of code inserted by the HPF compiler (runtime checks, calls to communication routines, etc.). Therefore, the loop order affects the run-time efficiency of the parallel program. Comparing the sequential runtimes and the one-processor runtimes of the XHPF generated parallel code, both fully optimized, we can interpret the difference as parallelization overhead. This overhead is significant; The ikj or kij variant require at least 26 processors to obtain the runtime of the optimized sequential version. On small machines with few processors it might even be impossible to reach the sequential time.

The loop partition match is illustrated in Table 2. The parallel code was executed on 32 processors of the Intel Paragon. The C and R extensions of the loop order triples mark type and location of interprocessor communication inserted by XHPF. C denotes preloop communication and R denotes a reduction add operation [10]. An additional concern is that for the ijk and ikj variants all communication is done in one operation. This indicates that one of the matrices is replicated on all processors, and the compiler reintroduces the scalability problem with respect to memory requirements that we tried to avoid by distributing all matrices. It is not clear whether this code is scalable.

The loop orders with the best timings in Tables 1 and 2 clearly illustrate the interaction between the optimization gap and loop partition match. The best loop order for the parallel code on one processor (jki) gives the worst runtime on 32 processors, whereas the best code on 32 processors (ijk) performs poorly on one processor. To investigate this effect, we measured a series of speedup curves for both the ijk and jki variants. The results are presented in Table 3.

The ijk variant scales well up approximately fifty processors, where the communication volume limits further speedup, independent of the optimization level. However, reasonable speedups are obtained with this variant. Superlinear speedups are caused by cache effects, not by swapping. With the problem size of 500×500, each processor can hold three matrices simultaneously. Each matrix requires 2 MB, whereas the main memory of one Paragon processor has 32 MB [1].

In contrast to the regular behavior of the ijk variant the speedup behavior of the jki variant

Number	*ijk* / `-O`		*ijk* / `-full`		*jki* / `-O`		*jki* / `-full`	
of procs	runtime	speedup	runtime	speedup	runtime	speedup	runtime	speedup
1	160.22	1.0	138.14	1.0	43.44	1.0	31.96	1.0
2	57.40	2.8	46.38	3.0	154.45	0.3	148.69	0.2
4	21.94	7.3	16.58	8.3	63.29	0.7	60.05	0.5
8	9.08	17.6	6.40	21.6	35.04	1.2	33.41	0.9
16	5.78	27.7	4.18	33.0	22.82	1.9	22.17	1.4
32	3.32	48.2	2.61	52.9	21.02	2.1	20.78	1.5
48	2.38	67.3	2.14	64.5	25.00	1.7	24.71	1.3
64	2.23	71.8	2.13	64.8	23.27	1.9	23.23	1.4
76	2.11	75.9	2.12	65.2	36.56	1.2	36.52	0.9
88	2.01	79.7	2.12	65.2	35.15	1.2	34.58	0.9
96	2.01	79.7	2.12	65.2	35.16	1.2	35.15	0.9

Table 3: Runtimes [seconds] and speedups of matrix-matrix multiplication (500×500) relative to the one-processor HPF code, *ijk*-variant and *jki*-variant with different optimization degrees. `-full` stands for `-O4 -Mvect -Mstreamall`.

Number	*ikj*		*jik*		*kij*		*kji*	
of procs	runtime	speedup	runtime	speedup	runtime	speedup	runtime	speedup
1	276.88	1.0	135.66	1.0	275.69	1.0	35.12	1.0
2	116.30	2.4	45.42	3.0	117.86	2.3	18.90	1.8
4	52.84	5.2	16.23	8.3	56.08	4.9	11.57	3.0
8	25.59	10.8	7.02	19.3	29.82	9.2	8.60	4.0
16	12.39	22.3	6.14	22.1	18.11	15.1	7.82	4.4
32	6.75	41.1	8.14	17.1	13.17	20.9	7.90	4.4
48	5.31	52.1	11.95	11.3	12.22	22.4	8.68	4.0
64	4.26	65.0	14.78	9.2	11.33	24.2	8.76	3.9
76	4.14	66.9	18.05	7.5	11.82	23.2	9.53	3.6
80	4.13	67.0	19.01	7.1	11.84	23.1	9.54	3.6
88	3.83	72.3	20.49	6.6	11.49	23.8	9.42	3.6
96	3.82	72.5	20.60	6.6	11.41	24.0	9.43	3.6

Table 4: Runtimes [seconds] and speedups of matrix-matrix multiplication (500×500) relative to the one-processor HPF code. All variants are compiled with full optimization.

is rather strange. The runtimes of the two-processor executions are about five times larger than those of the one-processor executions of the parallel code. We do not know why this happens only with the jki-variant. As shown in Table 4 below, no other variant exhibits this behavior. The lack of knowledge of the XHPF specific functions in the parallelized code makes the interpretation of the timings difficult. The only feasible way to analyze the peculiarities seems to be performance monitoring or empirical methods. Therefore, we present speedup curves for all other variants. We restrict these data to the versions with fully optimized native compilation, since the timing differences of the parallel codes with little and full optimization are not significant. The results are listed in Table 4. The runtime and speedup curves for the complete set of permutations are given in Figure 2.

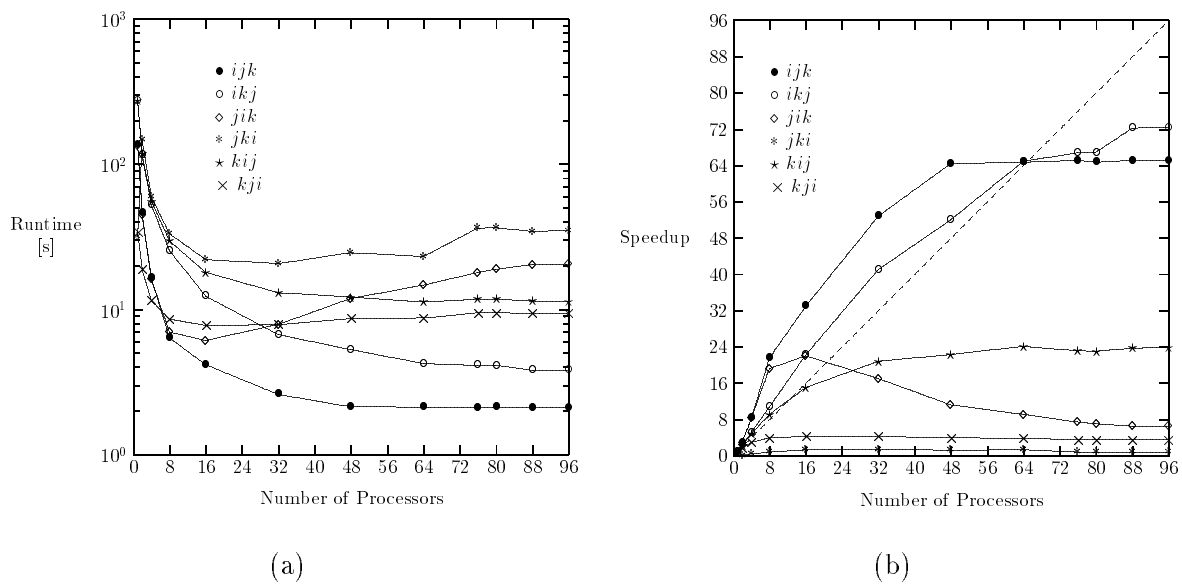

Figure 2: Runtime and speedup of dense matrix-matrix multiplication (500×500).

These experiments show that the loop partition match and the optimization gap strongly influence the parallel efficiency. Moreover, Figure 2 indicates that the optimal loop order depends on the number of processors. Hence, there is no variant that is optimal for all numbers of processors. Experimenting and profiling may be the only way to find the best variant for a particular number of processors, since it seems difficult to predict.

2.2 Gaussian Elimination

Our second example is the solution of a dense system of linear equations by Gaussian elimination with partial pivoting and backward substitution. In order to include the influence of partial pivoting on the performance we use an artificial, square system of order $1,000$ that yields 999 row exchanges (worst case).

To show the influence of the loop partition match and the optimization gap we use two sequential variants: a kij-ordered, forward-looking Gaussian elimination (gelim) combined with an ij backward substitution (backsb), and a kji-ordered, forward-looking Gaussian elimination with ji backward substitution [6]. The runtimes are listed in Table 5. They show that the

kji-elimination and ji-backward substitution lead to substantially better processor utilization, and that they have a higher potential for compiler optimizations. However, we will not further elaborate these aspects.

Optimization	gelim kij	backsb ij	gelim kji	backsb ji
-O	664.33	0.69	135.60	0.19
-O4 -Mvect	579.92	0.48	52.36	0.01
-Mstreamall	583.81	0.49	55.88	0.01

Table 5: Sequential runtimes [seconds] of Gaussian elimination with partial pivoting and backward substitution. The large variation in the runtimes show the sensitivity of the performance with respect to compiler optimizations and loop ordering.

The interesting issues here are the implementation decisions of the parallelizing compiler. In this respect the dense matrix-matrix product is straightforward compared to Gaussian elimination. For the matrix-matrix product the implementation follows immediately from the data distribution and the loop order, although this does not necessarily produce the desired efficiency. However, for the Gaussian elimination with partial pivoting several choices need to be made beyond the specification of the data distribution. These choices significantly affect the communication cost of the program.

Number	distribution only		forced optimization	
of procs	Elim	Back	Elim	Back
1	769.14	1.14	137.83	0.86
2	1599.49	2.06	95.84	2.40
4	2329.41	2.23	41.46	2.26
8	3955.12	3.05	32.90	3.06
16	7595.05	4.57	25.40	4.59
25	—	—	24.76	6.10
32	—	—	25.30	7.74
50	23096.44	10.51	28.76	10.79
64	—	—	32.56	13.43
80	—	—	37.15	16.22
96	—	—	41.83	19.04

Table 6: Runtimes [seconds] of Gaussian elimination and backward substitution for a nonsymmetric 1000×1000 matrix and 999 row exchanges due to partial pivoting. The runtimes show a large difference between the parallelization with only data distribution directives (distribution only) and with additional forced optimizations (forced optimizations). For the 'distribution only'-version we thought it was not useful to measure the runtimes for all numbers of processors.

We first consider the parallelization of the Gaussian elimination and backward substitution algorithms given in Appendix A. They are based on the sequential program without any algorithmic changes to assist an HPF compiler. We use a column cyclic distribution for the matrix, and we replicate the right hand side on all processors; the HPF distribution directives at lines 23 and 24 request this. Running the solver compiled with these directives only (without the

`capr$` directives) produces the runtimes given in Table 6 (distribution only), which are clearly unsatisfactory. Looking at the XHPF generated FORTRAN program discloses the problem; most communication is inserted inside the loops generating tremendous overhead due to a large number of communications and runtime checks. Our first work-around relies on XHPF specific forced optimizations (`ignore` directives) to move communication out of loops (line 35), to distribute a loop such that its body is executed on the processor that owns the requested elements (lines 34, 60, 76 and 81), and to prevent superfluous communication and runtime checks (lines 77 and 82). The runtimes of this version, given in Table 6 (forced optimization), are encouraging. However, this version still has some unnecessary communication overhead. The overhead occurs mainly because the scope of automatic parallelization (and optimization) is limited to single loops; optimizations over several consecutive loops are not considered. Also the reuse of data that have been fetched previously is ignored. Therefore, we have to indicate better implementations and optimizations to the compiler with a data-parallel programming style (see e.g. [7, 9]).

Before presenting our data-parallel programs, we discuss some design decisions for parallelization. Since we distribute the columns of the matrix cyclicly, the row exchange for pivoting can be done distributedly without communication. Furthermore, after the elimination factors have been computed and are available on all processors, the distributed update of the submatrix rows with the pivot row is also local. Therefore, communication is necessary only provide the index of the pivot row and the elimination factors to all processors. Hence, the pivot column (the pivot element and the elements below) and/or the elimination factors must be broadcasted from the processor that owns the pivot column to the other processors[1]. This leads to the following three parallelization variants for one iteration of the outer most loop (loop 200 in Appendix A):

1. The processor that owns the pivot column does the pivot search locally and broadcasts the result. After this, the row exchange is carried out distributedly. Then, the processor that owns the pivot column computes the elimination factors, and broadcasts the column which contains these. Afterwards, the update of the submatrix is carried out distributedly, and the update of the right hand side is replicated on all processors.

2. First, the pivot column is broadcasted, and the pivot search is replicated. Then the row exchange is carried out distributedly. The computation of the elimination factors is also replicated on all processors, and the update of the submatrix is carried out distributedly. Finally, the update of the right hand side is replicated on all processors.

3. The processor that owns the pivot column does the pivot search locally and broadcasts the result. After this, the row exchange is carried out distributedly. Then, the pivot column is broadcasted, and the computation of the elimination factors is replicated on all processors. Afterwards, the update of the submatrix is carried out distributedly, and then the update of the right hand side is replicated on all processors.

The three variants can be implemented with approximately the same efficiency, because they exhibit a similar amount of communication. The second variant should be the most efficient, because it needs slightly less communication than the others and requires the least synchronization. We consider two possible changes to the source program that create a more data-parallel programming style:

[1] It is also possible to distribute the pivot column and distribute the computation of the elimination factors. We will not consider this possibility because it will be very hard for the compiler to analyze whether this improves efficiency or not.

Local pivot search: We replace the scalars `ppiv` and `abspiv` by arrays of a size that equals the number of columns, and align these arrays with the columns of the matrix; see Appendix B. This way, for each column `col` of the matrix, we use the local vector elements `ppiv(col)` and `abspiv(col)` in the pivot search instead of (replicated) scalars that incur communication inside the loop.[2] Now, only the final result must be communicated to the other processors. This implements the first scheme.

All local: We declare a replicated (dummy or automatic) array (`pcol`) of the size of a matrix column, and copy the pivot column into that array; see Appendix C, loop 400. This copying leads to the broadcast of the pivot column. Furthermore, we replace all references to the pivot column by corresponding references to the replicated array (`pcol`). This leads to strictly local work for the pivot search, the row exchange, the computation of the elimination factors, and the update of the rows and the right hand side. This implements the second scheme.

The *local pivot search*-version without any forced optimizations does not have the desired effect. The compiler still broadcasts all intermediate results of the distributed array elements `ppiv(piv)` and `abspiv(piv)` in the pivot search. This is strange because these elements are local to the processor that executes the loop and (obviously) only the values after the loop are important for the rest of the algorithm and the other processors. The communication inside the loop can be avoided with the XHPF parallelizer by using a forced optimization that inhibits communication. However, without any further warning this leads to a segmentation violation in one of the APR runtime library routines if the program is executed on more than one processor. With several other 'forced optimizations' and algorithmic changes we could get the program to run, but we never obtained reasonable timings for the *local pivot search*-version.

Also the *all local*-version does not lead to the desired implementation immediately; the compiler still creates (now obviously) unnecessary communication related to the use of the replicated vector. This is the more disturbing since it is clear from the program that no such communication is necessary, and the program entirely follows the data-parallel paradigm without any need for compiler optimizations. First of all if the array `pcol` is declared as an automatic array inside the subroutine `gelim` the compiler implements the copying of the pivot vector by broadcasting each element separately instead of broadcasting the whole vector once. Furthermore, it puts a subroutine call for computing indices for the array `pcol` inside the loop. This leads to a very poor implementation; see the runtimes in Table 7 in the 'automatic array' column. However, if the array `pcol` is passed as a dummy array to `gelim` the pivot column is broadcasted as a whole, and the index calculation is moved out of the loop. This leads to the timings given in Table 7 in the 'dummy array' column. This type of performance differences for slightly different program versions is, of course, very undesirable. Especially, since from a programming point of view the 'automatic array' version is better, because the array `pcol` has no meaning outside the subroutine `gelim`.

The use of forced optimizations is undesirable for the reason already mentioned in the introduction. However, there is one more potential problem, which also plays a role if the compiler forces the programmer to make meaningless changes to the program only to prevent poor implementations. If the compiler insists on generating an inefficient program, the programmer is prone to resort to 'non-constructive programming'. That is, instead of carefully designing a program that assists the compiler to find the most efficient implementation, the programmer will

[2] According to the data-parallel programming paradigm scalars are replicated on all processors and therefore are broadcasted if they are updated on some processor.

Number of procs	automatic array		dummy array	
	Elim	Back	Elim	Back
1	326.06	1.06	136.81	1.07
2	703.54	1.17	72.02	1.15
4	1297.54	1.34	39.81	1.36
8	3009.48	1.75	24.06	1.78
16	7203.39	2.49	17.50	2.52
25	—	—	16.20	3.31
32	—	—	16.31	3.69
50	23096.44	10.51	18.58	5.04
64	—	—	20.31	5.99
80	—	—	23.28	7.13
96	—	—	26.59	8.18

Table 7: Runtimes [seconds] of data-parallel style Gaussian elimination with partial pivoting and backward substitution, *all local*-version.

leave a (possibly poor) program for what it is, and prevent inefficient parallel implementations by inserting a large number of forced optimizations in the program. This leads to unreliable code, which is based on numerous unclear assumptions.

After the Gaussian elimination we compute the solution by backward substitution. For the given distribution of the matrix there are two straightforward implementions: a row-oriented and a column-oriented backward substitution with the solution vector `x` aligned with the columns of the matrix. The column-oriented version and the row-oriented version are given in Appendix D. It is clear from these programs that the column-oriented version requires more communication than the row-oriented version. The column-oriented version needs a broadcast of the array element `x(col)` and a one-to-all scatter of `a(1:row-1,col)` in each step, whereas the row-oriented version needs only a reduction add over the local products `a(row,col)*x(col), col=row+1, ... ,n` (cf. line 31 in the column-oriented version and line 29 in the row-oriented version), which costs about the same (in communication) as the broadcast of a scalar. Therefore, we have chosen the row-oriented version for our timings.

2.3 Simple Flow Application

The third example involves the finite volume discretization of a simple flow problem on a regular grid and a fixed number of steps of the nested iterative (linear) solver described in [13].

In a large application, a single array may be passed to and updated by many different subroutines with different access patterns. Conversely, a single subroutine may be called at many places in the program with array arguments that differ in size, shape, and/or distribution. The complexity of the entire set of potential optimizations that results from the interprocedural analysis may prevent the compiler from making the appropriate optimizations, leading to sequential execution of, in principal, parallel parts of the program, or to excessive communication and synchronization. None of the specific parts of the program at hand plays a role by itself; indeed, each of the routines used in this program parallelizes well by itself. It is the global parallelization and the propagation of dependencies that causes the problems described here.

Regarding these problems we will first describe some concepts that underlie the paralleliza-

tion and distribution of loops and arrays. Different compilers will address these concepts in different ways. However, since the distribution of arrays and parallelization of loops lie at the heart of data-parallelism the concepts themselves must be addressed in some way. Based on this introduction we will then describe the problems we found for our particular compiler.

With each array we can associate an index set that consists of the valid index references to that array. With each loop we can associate an index set that consists of the values that are assigned to the loop index during the execution of the loop. The distribution of an array `a` is described by a function from the array index set into a processor index set P, $M_{I_a,P} : I_a \rightarrow P$.[3] Likewise, we can describe the distribution of a loop `l` by a function from the loop index set to a processor index set P, $M_{I_l,P} : I_l \rightarrow P$. We assume that all statements within one iteration of the loop are carried out on the same processor. This is the case for the APR XHPF compiler and for example in the Oxygen compiler [12], however it is not defined in the HPF language specification [8]. Through this use of index sets the distribution of data and partitioning of loops are described in the same way.

The programmer can indicate to the compiler that the distribution of the elements of one array should depend on the distribution of the elements of another array through the alignment directive [8]. This directive describes a linear function from the index set of one array, the **alignee**, into the index set of another array, the **align target**. Given the arrays `a` and `b` with the index sets I_a and I_b, the alignment $\alpha : I_a \rightarrow I_b$, a processor set P, and the 'distribution' $M_{I_b,P}$ of the array `b`, we can describe the interpretation of the alignment indicated by α. For the processor set P this alignment leads to the 'distribution'

$$M_{I_a,P} : I_a \rightarrow P \quad \text{such that} \quad M_{I_a,P}(i) = M_{I_b,P}(\alpha(i)).$$

In order to minimize data movement for the execution of a distributed loop an optimizing compiler will try to assign the execution of each loop iteration to the processor that owns the array elements referenced in that loop iteration. This implies functions from the loop index set to the array index sets of the arrays that are referenced inside the loop, which leads to certain desired 'alignments' between the loop index set and the array index sets. In fact, for the APR XHPF compiler the alignment of the loop index set with one array index set can be indicated explicitly through a directive.

In this article we will refer to alignments that are indicated explicitly by directives as **explicit alignments**. We will refer to all other 'alignments' that an optimizing compiler may make to reduce communication as **implicit alignments**. Such implicit alignments come, for example, from references to different distributed arrays in a single distributed loop. Several partitioning strategies are used in data-parallel compilers that lead to different implicit alignments, e.g. *owner computes* and *almost owner computes* [11]. However, they all suffer from serious limitations with respect to optimization when applied rigorously, so in practice relaxed versions are used. APR uses a so-called *owner sets* strategy [5].

We illustrate the effects of partitioning by means of an example. Consider a loop in which two arrays are referenced. The loop index set is given by I_l, the index sets of the arrays are I_a and I_b, and the alignments of the loop with the array index sets are given by the functions $\lambda_a : I_l \rightarrow I_a$ and $\lambda_b : I_l \rightarrow I_b$. In order to keep all array references local in this loop for all possible P, the distributions $M_{I_a,P} : I_a \rightarrow P$ and $M_{I_b,P} : I_b \rightarrow P$ must fulfil, for all possible P,

$$\forall i \in I_l : M_{I_a,P}(\lambda_a(i)) = M_{I_b,P}(\lambda_b(i)).$$

[3] We use the term processor for simplicity; one may use instead process, thread, and so on. These have to be mapped to physical processors in turn.

This defines an *implicit alignment* of I_a and I_b in the following sense,

$$\exists \alpha_b : \lambda_a(I_l) \rightarrow I_b \quad \text{such that} \quad \alpha_b(\lambda_a(i)) = \lambda_b(i)$$

or

$$\exists \beta_a : \lambda_b(I_l) \rightarrow I_a \quad \text{such that} \quad \beta_a(\lambda_b(i)) = \lambda_a(i)$$

Obviously *implicit alignments* can have a much more complicated form than *explicit alignments*, and they may be only a relation on part of the index set of the alignee. Implicit alignments may also arise indirectly by aligning several arrays explicitly with the same array, or through an explicit alignment in combination with a distributed loop referencing more than one distributed array. Although implicit alignments indicate potential compiler optimizations, they might create problems as well, in particular at subroutine boundaries, as we will discuss below.

We will now describe four problem classes that ocurred in the parallelization of our application.

- **Directive-based implicit alignment**

  Subroutines may declare the alignment and distribution of dummy arguments. These declarations can lead to implicit alignments of actual arguments of the subroutine in separate calls. Consider a scenario where subroutine `sub(x,y)` aligns array `x` with array `y` on subroutine entry by means of an `align` directive. Then, two calls to this subroutine of the form `call sub(a,b)` and `call sub(c,b)` lead to an implicit alignment of the arrays `a` and `c`. The compiler may want to propagate the two alignments to the calling routine to prevent redistribution inside the subroutine (like the APR XHPF compiler does). This will only be possible if the potential distributions (indicated by the programmer, for example) of the index sets of `a` and `c` satisfy the requirements from this implicit alignment.

  We call the implicit alignment of arrays that arises in this way **directive-based implicit alignment**, because it arises out of alignments that are indicated by a directive.

- **Loop-based implicit alignment**

  As we described above, for a distributed loop containing references to a distributed array an optimizing compiler will try to align the loop index set with the index set of the distributed array such as to minimize communication. We refer to this type of implicit alignment as **loop-based implicit alignment**. If such a loop occurs inside a subroutine and the distributed array is a dummy argument, this leads to the implicit alignment of the loop index set with the index set of the actual argument in each call to this subroutine. If the distribution of the loop index set could be different for each call to the subroutine, this would not create any problems. However, this seems generally not to be the case. Except for the specific processor set, which is only known runtime (at loading), the distribution mapping will be fixed by the compiler, because it typically generates only one object module for the subroutine. This holds, for example, for the APR XHPF compiler and for the ADAPTOR-tool [3]. A fixed distribution mapping of the loop index set, however, leads to the implicit alignment of all the actual arguments in separate calls to the subroutine. The effect is the same as if all the actual arguments were referenced in this loop at the same time.

- **Aliasing**

  If more than one symbolic name to a reference (variable or memory location) exists within a loop, we call this an **alias**, following the APR XHPF documentation [5]. Aliasing creates problems for parallelizing a loop if the name is assigned a value within the loop.

In practice, the compiler may also spot many 'potential' aliases that are not real. This depends largely on the (interprocedural) analysis capabilities of the compiler. Fake aliases lead to parallelization and optimization problems that require forced optimizations.

- **Interprocedural propagation**

  Because of the interprocedural analysis, parallelization problems may propagate through the entire program. A problem that comes up in one subroutine may prevent some array from being distributed, which leads to problems in other subroutines, and so on. The interprocedural analysis may have the unwanted side effect of globalizing local problems.

It should be clear that implicit alignments are not in themselves a problem. They arise naturally from the program and form a source for compiler optimizations. However, where implicit alignments lead to conflicts or potential conflicts, the analysis and optimizations capabilities of the compiler are put to the test to produce reasonable choices. Especially in large programs the number and complexity of implicit alignments may become hard for the compiler to deal with. Compilers, therefore, tend to make several passes through the program making incremental improvements or adding optimization information in a database (the APR XHPF compiler) or in the program itself (see [12]) that may lead to further optimizations in a future pass of the compiler. However, there is no guarantee that such an incremental optimization strategy leads to good optimizations; this probably depends largely on how compilers deal with choices made in previous passes. The final results of the compiler analysis are very important, since for reasons of safety the compiler will sequentialize, either explicitly or implicitly through synchronization or communication in a loop, any part of the code where a correct parallel execution is not guaranteed. On the other hand, unwanted implicit alignments are often the result of poor programming, at least from a data-parallel point of view.

In the following, we will illustrate the four problem classes by examples that arose from of the parallelization of our application program.

Directive-based implicit alignment

First, we consider potential conflicts arising from directive-based implicit alignment, and then we consider a distribution problem associated with workspace arrays. Consider a simplified version of the subroutine `daxpy`[4] (vector update) that is invoked twice within another routine. See the program fragment given in Figure 3.

The explicit alignment of dummy arguments in the `daxpy` subroutine and the two calls in the program lead to an implicit alignment of the arrays `a` and `c`. The `daxpy` can be executed without any communication if the arguments have the same distribution. The actual distribution of the arguments does not matter, as long as it is the same for both arrays. In the program in Figure 3, however, the implicit alignment leads to a problem regarding the distribution of the array `b`, and it depends on the rest of the program what the best choice will be for the distribution of `b`. Although for this example the conflict seems the programmer's fault, one should realize that the choice of distribution for `c` could have arised from previous optimizations of the compiler (without the explicit directive). A conflict like this does not only arise with different distributions, but also for arrays with the same distribution but with different size or shape, and so on. In the small application at hand we had to resolve such problems several times.

[4]The `daxpy` routine as it is defined in the BLAS has different increments for the two vectors `dx` and `dy`; this makes alignment impossible in the general case.

```
1        subroutine daxpy(n,da,dx,dy)
2  c
3  c     compute dy = da*dx + dy
4  c     da is a scalar, dx and dy are vectors
5  c
6        double precision dx(n), dy(n)
7        double precision da
8  chpf$ align dx(k) with dy(k)
9  c
10       do i = 1, n
11          dy(i) = dy(i) + da*dx(i)
12       end do
13       end
14 c
15       program use_daxpy
16       double precision a(n), b(n), c(n)
17       double precision alpha, beta
   chpf$ distribute a(block)
   chpf$ distribute c(cyclic)

            ⋮

18       call daxpy(n,alpha,a,b)
19       call daxpy(n,beta,b,c)

            ⋮

20       end
```

Figure 3: The `daxpy` routine and its use lead to directive-based implicit alignment

A problem that will arise frequently in converting existing F77 programs to HPF, and which can easily be avoided, is the implicit alignment of unrelated arrays due to the use of workspace arrays passed to a subroutine. Consider the program fragment in Figure 4. The programmer wants array `a` to be block distributed (line 10) and the array `b` to be cyclicly distributed (line 11). If the parallelizing compiler obeys these directives, any distribution for array `h` will lead to large communication overhead. Typically, the parallel program will redistribute `h` upon entry to the subprogram.

The alignment directive does not solve the problem; it only indicates to the compiler that it is probably better to align (redistribute) the array `dummy_vector` at the start of the subroutine than to do the communication in the subroutine element-wise. The solution is to use the dynamic allocation features of HPF (derived from F90); that is to use automatic arrays as workspace.

2.3.1 Loop-based implicit alignment

Second, we look at a problem associated with loop-based implicit alignment. In the `daxpy` routine given in Figure 3 we replace the prescriptive (desired) alignment [8] at line 8 by a descriptive (asserted) alignment [8]. The descriptive alignment (`chpf$ align dx with *dy`) asserts to the compiler that the actual arguments are (already) aligned on entry to the subroutine. Now, consider the use of the `daxpy` in the program fragment of Figure 5.

```
1        subroutine sub1(n,vector,dummy_vector)
2  c     dummy_vector is a workarray
3  chpf$ align dummy_vector with vector
4  c
5        forall (i=1:n) (dummy_vector(i)=vector(i))

            ⋮

6        end
7  c
8        program
9        double precision a(n), b(n), h(n)
10 chpf$ distribute a(block)
11 chpf$ distribute b(cyclic)

            ⋮

12       call sub1(n,a,h)
13       call sub1(n,b,h)

            ⋮

14       end
```

Figure 4: Example program illustrating unnecessary implicit alignment

For each call to `daxpy` the compiler knows that the actual arguments are aligned and that no communication is necessary. However, the `daxpy`-loop from the first call (line 7) should be distributed due to the block-distribution directive in line 5, whereas the `daxpy`-loop from the second call (line 8) should be replicated due to the replicated-distribution directive in line 6.

The APR XHPF compiler generates a single implementation of the `daxpy` routine. This results in the compiler decision to either replicate the loop, which gives (tremendous) communication and calculation overhead for the distributed arrays, or to parallelize the loop, which gives a tremendous amount of communication overhead for the replicated arrays. Here, the compiler treats unrelated arrays that are arguments in different calls of the subroutine as if they were in the same loop; this causes the implicit alignment of `a`, `b`, `c`, and `d`. In our case we needed forced optimizations or two different `daxpy` subroutines, one for block-distributed array arguments and one for replicated array arguments, to obtain an efficient parallel code.

It is interesting to note that if the `daxpy` program were not in a subroutine but in-lined by the programmer there would be no problem at all. The compiler could have implemented the optimal choice for each loop. For the implementation of a simple routine like the `daxpy` this may be a good idea; the entire (simplified) `daxpy` can be implemented in a single forall statement. However, the same problem also appears for more complicated kernel routines that are not suited for in-lining at the source code level.

It is not defined in the HPF standard that a subroutine will have only one object module for all invocations. However, that seems to be the standard implementation. Having different subroutine sources for different argument distributions, alignments, or shapes is undesirable because it means that the programmer has to keep track of the different uses, and he has to find out the different uses for an existing program. This is all the worse or even impossible since

```
1        program loop_align
2        double precision a(n), b(n)
3        double precision c(m), d(m)
4  c
5  chpf$ distribute (block) :: a, b
6  chpf$ distribute (*)     :: c, d

            ⋮

7        call daxpy(n,alpha,a,b)
8        call daxpy(m,beta ,c,d)

            ⋮

9        end
```

Figure 5: The `daxpy` routine and its use lead to undesired loop-based implicitly alignment

the distributions of arrays may not be known before the program is compiled (the compiler may choose other distributions than indicated in the program) or even not before runtime.

2.3.2 Aliasing

Third, we consider parallelization and optimization problems that arise from aliases. In F77 we often address parts of arrays as independent items. For example, we may consider a two dimensional array as a matrix and access its 'columns', or we may consider a three-dimensional array as a three-dimensional grid of which we access planes or lines (for example in a three-dimensional FFT). If we pass two (or more) different parts of a distributed array to a subroutine and modify at least one part in a partitioned loop, potential aliases occur and the compiler has to determine whether they are real or not. If the compiler cannot establish that no alias exists, it will typically execute the loop sequentially. Alias problems arise in the generation of orthogonal bases in the iterative solver. A two-dimensional array is used to represent a sequence of vectors; these vectors are generated by multiplying the last generated vector with a matrix and then orthogonalizing it on the previously generated vectors (inner products and vector updates); see Figure 6. In the subroutines for the matrix-vector product and the vector update two (disjunct) parts of the same array are used and one is updated. If the compiler cannot check that these parts are disjunct, it cannot parallelize the loops in these subroutines. Using dynamic allocation, we made an implementation of the program that enables the compiler to check that the referenced parts of the array are always disjunct; however, the compiler still refused to parallelize the routines. We could solve the problem only with forced optimizations.

An alternative implementation in (full) HPF may be to represent the sequence of vectors by an array of structures each of which contains a pointer to a vector. However, pointers are not part of the HPF subset and the construction is rather cumbersome. Moreover, this approach does not provide a solution if it is necessary to traverse a higher dimensional array in more than one way. Consider, for example, a 3D FFT over a three-dimensional array. We would like to consider such an array first as a collection of x-vectors, then y-vectors, and finally z-vectors. This cannot be done with pointers in HPF.

```
1        subroutine gmreso(n,vv,...)
2  c
3        double precision r(n), vv(n,m+1)
4        double precision hh(m+1,m)
5  c
6  chpf$ distribute r(block)
7  chpf$ align vv(i,*) with r(i)
8  chpf$ distribute (*,*) hh
9  c
```

⋮

```
10       ***** Arnoldi *****
11       do i = 1, n
12          vv(i,1) = r(i)/res\_norm
13       end do
14 c
15       do i = 1, m
16 c        ***** v_{i+1} = Av_i *****
17          call matvec(n,vv(1,i),vv(1,i+1),...)
```

⋮

```
18          do j = 1, i
19 c           ***** h_{ij} = < v_j , v_{i+1} > *****
20             hh(j,i) = ddot(n,vv(1,j),1,vv(1,i+1),1)
21 c           ***** v_{i+1} = v_{i+1} - h_{ij}*v_j *****
22             call daxpy(n,-hh(j,i),vv(1,j),1,vv(1,i+1),1)
23          end do
24       end do
```

⋮

```
25       end
```

Figure 6: Parallelization problems due to aliases created by passing parts of `vv` as different vectors.

2.3.3 Interprocedural Propagation

Fourth, we look at the influence of parallelization problems in one subroutine on the rest of the program. Because of the interprocedural analysis, parallelization problems may propagate through the entire program. Consider the program fragment in Appendix E. Potential aliases for the different parts (vectors) of the array `vv` in the Arnoldi part in the subroutine `gmreso()` prevent the compiler from parallelizing the matrix-vector product and the `daxpy`. This in turn seems to prevent the distribution of arrays `uu` and `cc` and creates huge communication overhead. The resulting replication of the arrays `uu` and `cc` gives rise to problems parallelizing other loops, and so on. Finally, significant parts of the code are not parallelized, or suffer from excessive overhead and communication. The propagation has the additional problem for the programmer that it is not always clear where the problem starts, and how it can be solved or at least limited to a small part of the program. The interprocedural analysis has the unwanted effect of making local problems global.

2.3.4 Experimental Results

The runtimes for the discretization and a fixed number of steps of the iterative solver in the final HPF program are given in Table 8 for three problem sizes and several numbers of processors. For one processor the table gives the runtime for the original sequential code (seq) and for the HPF code (hpf) executed on one processor. For the first two problem sizes the program fits in the memory of a single processor, for the largest problem size the program does not fit in the memory of a single processor (so some paging is done) but fits in memory for two processors or more. So, considering the size of the machine, 96 processors, this is still a small problem.

It is clear from a comparison between the sequential runtimes of the original program and the HPF program that the latter induces a substantial amount of overhead. Since no special measures are taken in the solver to improve the effects of a large number of inner products, which need global communication, the speedup deteriorates rapidly for smaller problems [14, 15]. For the largest problem the speedup on 32 processors is approximately eight. Moreover, if we look at a specific number of processors (say 64), then we see that the runtime increases by only 25% to 35%, whereas the problem size increases by a factor of 2.5. So for very large problems the overhead induced by the compiler becomes less noticeable, and the (relative) performance improves.

3 Recommendations and Discussion

In this section we discuss some of the previously described problems and their proposed solution, and we propose an order in which the steps in converting a program from F77 to HPF should be applied.

The recommendations are mainly based on the work with the APR XHPF compiler, although we have tried to make them general by considering the background of the problems. Furthermore, we encountered several problems that seemed typical for the APR compiler, and therefore we did not discuss them in detail, even though some of them caused severe performance problems. It is likely that other compilers will have similar idiosyncratic problems.

Another concern is that only an HPF language specification has been defined, and this specification (of course) makes no requirements on the implementation of an HPF-program, nor does it require a minimal set of optimizations that the HPF programmer can assume. Indeed, at the main program level all directives (particularly distributions and alignments) are

Number of procs	$4 * 10^4$		$1 * 10^5$		$2.5 * 10^5$	
	discr.	it.solver	discr.	it.solver	discr.	it.solver
1 (seq)	0.38	5.20	1.00	13.24	2.38	42.38
1 (hpf)	0.74	15.23	1.84	37.19	4.58	112.78
2	0.48	8.24	1.11	19.23	2.71	81.81
4	0.23	4.69	0.54	10.30	1.35	66.65
8	0.19	3.12	0.37	6.61	0.65	14.47
16	0.14	2.61	0.22	4.06	0.42	8.36
32	0.09	2.69	0.17	3.49	0.20	5.36
64	0.14	3.69	0.18	4.26	0.21	5.32
96	0.18	4.78	0.18	5.16	0.22	5.83

Table 8: Runtimes [seconds] for the finite volume discretization and 25 iterations in total of the nested iterative solver for three problem sizes: the sequential program and the HPF program for several numbers of processors.

so-called *prescriptive*; that is, they indicate a desired feature but they may be ignored. Strictly speaking, the programmer has no control over the actual distribution, although in practice the situation will not be so bad. Nevertheless, any distributions that the compiler will use or 'grant' the programmer and any optimizations it will make depend on the analysis capabilities of the compiler. This severely hampers the possibility to make assumptions about the generated code. Moreover, the wide range of possible implementations by compiler developers poses a threat to the aim of *performance portability* that is mentioned in section 2.2 of the HPF Language Specification [8]. Preliminary testing of our programs using the PGI[5] compiler confirmed our fears in this respect, especially since other people report quite favourable results for this compiler.

As an example of the range of possible implementations consider the following case. In a subroutine the programmer asserts the alignment of two dummy arguments (a descriptive alignment). This means that it is the programmer's responsibility that this alignment is in fact true upon each entry to the subroutine. However, in the main program the programmer cannot insist on any particular distribution, because he can use only prescriptive directives, which may be ignored by the compiler. Recognizing this problem the HPF Language Specification [8] states

> All this[6] is under the assumption that the language processor has observed all other directives. While a conforming HPF language processor is not required to obey mapping directives, it should handle descriptive directives with the understanding that their implied assertions are relative to this assumption.

We can immediately distinguish two ways of handling the problem that are at opposite ends of a wide range of possibilities. One way is that as soon as the compiler does not observe a single directive or encounters a single analysis problem (and its analysis capabilities may be poor) it will simply ignore all descriptive directives. The other way is that the compiler does extensive interprocedural analysis and *propagates* such requirements on dummy arguments to the actual arguments in the calling routine(s), and so on. In this way the compiler can check whether decisions it made elsewhere in the program have an influence on this particular descriptive directive. In fact, the directive may even influence distribution decisions taken at higher levels

[5] Trademark of Portland Group Inc.

[6] It is the programmer's responsibility that the asserted statement holds.

in the program. Obviously, there are many possibilities in between the two extremes. All are influenced by the quality of the analysis and the type of analysis being done, and by the optimizations that the compiler supports.

In order to support novice users of HPF we provide a small summary of those points of the parallelization process that appear most important to us.

One should always start with a sound analysis of the program, the algorithms involved, and the purpose of the program.[7] From this analysis a parallelization strategy should be developed that involves the distribution of both data and work. The distribution of data is mainly done through declarations and directives (distribute and align). The distribution of work (in HPF) is done by using a data parallel programming style and the use of statements and constructs that indicate independence of loop iterations (forall and independent), and by using the HPF library (not in the HPF subset). As far as optimized functions are available from the HPF library they should be used. In order to assist the compiler in generating an efficient parallel program, one should reduce the complexity of nested loop structures and index expressions as much as possible. Furthermore, declaring arrays only where they are needed and using the dynamic allocation features generally improves the generated parallel code, but one should be aware that sometimes it is better to declare large, distributed arrays at a higher level in the program to facilitate the alignment with other arrays.

In order to take care of the loop partition match, one should make loop nestings and array distributions such that communication can be moved out of the loops as much as possible. Less subroutine calls for communication and runtime checks inside the loops will also increase the optimization opportunities for the native compiler, which may reduce the optimization gap. The optimization gap may be further reduced by adjusting the loop order in the (original) HPF program such as to improve locality, vector length, or other features in the program generated by the HPF compiler. However, one should realize that such changes may improve the performance on one parallel computer and reduce it on another. Furthermore, in order to do this well a thorough understanding of the HPF compiler itself is necessary. In many cases loop partition match and optimization gap requirements will conflict. One possibility to resolve this is to change the algorithm and/or the data structures; in numerical simulations often different algorithms can be used to obtain the same result, and the choice for the current algorithm may have been based on assumptions that do not hold in a a parallel (or HPF) setting. Another, but more machine specific, consideration is that on a machine with few powerful processors optimizations by the native compiler might be much more important than the cost of communication, whereas on a machine with many relatively slow processors this might be the opposite.

Changing the algorithms and data structures or replacing one algorithm by another should always be tried to reduce problems if simple 'tricks' do not work.

To prevent problems with implicit alignments one should be careful with directives for dummy (array) arguments in subroutines and with the actual arguments of the subroutines calls. Often alignments can be made at a higher level to prevent some of the implicit alignment problems. This assumes a certain amount of interprocedural analysis though. It is important to be aware of what happens at subroutine boundaries. Furthermore, one should consider the inlining of small often used subroutines. Another possibility is to have different versions of certain subroutines for arguments with different sizes and/or distributions. If possible one should prevent loops referencing arrays that cannot be aligned.

Alias problems can sometimes be prevented by using automatic arrays; one should not pass

[7] We hope that this sounds like an obvious truth to everybody. We mention it because we feel that recently HPF has been oversold to be very easy to use, and to involve nothing more than inserting a few directives at the start of a program or subroutine. This, however, is not our experience.

workspace arrays. Another possibility is to use pointers, although this only helps in simple cases. Finally, one may use forced optimizations, if changing the algorithms and data structures does not work.

To avoid interprocedural propagation problems, one should realize that the best way to avoid them is by removing the problem that is the source of the propagation. To remove remaining problems one should again consider changes to algorithms and/or data structures. Finally, one could use forced optimizations.

Only at the end one should consider the use of forced optimizations to improve the efficiency of the parallel code by removing unnecessary communication and runtime checks.

4 Conclusions

We have illustrated several problems, the loop partition match, the optimization gap, problems with program sections whose parallelization is not clear from the data distribution, implicit alignment and alias problems, and problems introduced by the interprocedural analysis. Although we have investigated these problems only for the APR XHPF compiler, we think they play a role for other compilers as well. For most of these problems we have indicated possible solutions at the program level (even though they may not work work with the XHPF compiler). However, the solution of these problems at compiler level would substantially enhance the performance of the parallel program and substantially reduce the effort of the programmer. We obtained reasonable performance for our programs, but it took some program changes or algorithmic changes, and several APR specific forced optimizations.

Concerning the APR XHPF compiler more specifically, we can draw the following conclusions. The program must be (re)written in a data-parallel programming style. Furthermore, a rather detailed understanding of the compiler is necessary to obtain good results, and the compiler is sometimes over-conservative (which probably indicates insufficient analysis capabilities), so that considerable time must be devoted to removing unnecessary communication calls and runtime checks.

References

[1] Arbenz P., *First Experience with the Intel Paragon*, in 16th Speedup Workshop on Vector and Parallel Computing, Speedup **8**(2), pages 53–58, December 1994.

[2] Bailey D. H., *RISC Microprocessors and Scientific Computing*, in Supercomputing'93, Portland, Oregon, pages 645–654, November 1993.

[3] Brandes T. and Zimmermann F., *Data Parallel Programming on IBM's Scalable Parallel Systems with the ADAPTOR Tool*, 1994 (URL `http://www.gmd.de/SCAI/lab/adaptor/documents.html`).

[4] Crooks P. and Perrott R., *Language Constructs for Data Partitioning and Distribution*, Technical Report, Department of Computer Science, Queen's University of Belfast, 1994.

[5] *FORGE HPF Parallelizer XHPF User's Guide 1.0*, Applied Parallel Research, 550 Main Street, Suite I, Placerville, CA 95667, 1993.

[6] Freeman T. L. and Phillips C., *Parallel Numerical Algorithms* (Prentice Hall, Englewood Cliffs, 1992).

[7] Hatcher P. J. and Quinn M. J., *Data-Parallel Programming on MIMD Computers* (MIT Press, Cambridge, 1991).

[8] *High Performance Fortran Language Specification*, High Performance Fortran Forum, November 1994.

[9] Hillis W. D. and Steele G. L., Jr., *Data Parallel Algorithms*, Communications of the ACM **29**(12), pages 1170–1183, December 1986.

[10] Kumar V. et al., *Introduction to parallel computing: Design and Analysis of Algorithms* (Benjamin/Cummings, Redwood City, 1994).

[11] Ponnusamy R. and Saltz J. and Choudhary A., "Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse," Technical Report **UMIACS-TR-93-32.1**, University of Maryland, October 1993.

[12] Rühl R., *A parallelizing compiler for distributed memory parallel processors*, PhD thesis, ETH Zürich, 1992.

[13] Sturler De E., *Nested Krylov methods based on GCR*, Technical Report **93-50**, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft, The Netherlands, 1993. (accepted for publication in the Journal of Comp. and Appl. Mathematics, North-Holland, Amsterdam).

[14] Sturler De E. and Vorst Van der H. A., *Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers*, Technical Report **832**, Mathematical Institute, University of Utrecht, Utrecht, The Netherlands, 1993. (accepted for publication in Applied Numerical Mathematics).

[15] Sturler De E. and Vorst Van der H. A., "Communication cost reduction for Krylov methods on parallel computers," In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking*, Lecture Notes in Computer Science 797, pages 190–195, Berlin, Heidelberg, Germany, 1994. Springer-Verlag.

[16] Zima H. and Chapman B., *Supercompilers for Parallel and Vector Computers*, ACM Press Frontier Series (Addison-Wesley, Wokingham, 1991).

A Linear Equation Solver

This is the parallelization of the sequential implementation; see Table 6 for results.

```
c
c     Gaussian elimination with partial pivoting
c
      subroutine gelim(a,lda,b,n,tol,perm,errflg)
c
c     *** in/out variables ***
c
      double precision  a(lda,n), b(n), tol
      integer           lda, n, perm(n), errflg
c
c     *** local variables ***
c
```

```
13        integer           row, col, piv, ppiv, itmp
14        double precision  abspiv, dtmp
15  c
16  c     *** intrinsic functions ***
17  c
18        intrinsic abs
19        double precision  abs
20  c
21  c     *** distribution and alignment ***
22  c
23  chpf$ distribute a(*,cyclic)
24  chpf$ distribute b(*)
25  chpf$ distribute perm(*)
26  c
27        errflg = 0
28        do 200 piv = 1, n-1
29  c
30  c        *** pivot search ***
31  c
32           ppiv = piv
33           abspiv = abs(a(piv,piv))
34  capr$    do par on a<:,piv>
35  capr$    ignore sync com
36           do 110 row = piv+1, n
37              dtmp = abs(a(row,piv))
38              if (dtmp .gt. abspiv) then
39                 abspiv = dtmp
40                 ppiv   = row
41              endif
42   110     continue
43  c
44  c        *** permutation for perm ***
45  c
46           itmp       = perm(piv)
47           perm(piv)  = perm(ppiv)
48           perm(ppiv) = itmp
49  c
50  c        *** check for singularity ***
51  c
52           if (abspiv .lt. tol) then
53              errflg = ppiv
54              write (*,*) 'tolerance exceeded: ', abspiv
55              return
56           else if (ppiv .ne. piv) then
57  c
58  c           *** permutation for a ***
59  c
60  capr$       do par on a<:,col>
61              do 120 col = 1, n
62                 dtmp        = a(piv,col)
63                 a(piv,col)  = a(ppiv,col)
64                 a(ppiv,col) = dtmp
65   120        continue
66  c
```

```
c           *** permutation for b ***
c
            dtmp    = b(piv)
            b(piv)  = b(ppiv)
            b(ppiv) = dtmp
         endif
c
c        *** elimination ***
c
capr$    do par on a<:,piv>
capr$    ignore all com
         do 125 row = piv+1, n
            a(row,piv) = a(row,piv) / a(piv,piv)
 125     continue
capr$    do par on a<:, piv+1~1>
capr$    ignore all com on a<1+piv~1,piv>
         do 140 col = piv+1, n
            do 130 row = piv+1, n
               a(row,col) = a(row,col) - a(row,piv) * a(piv,col)
 130        continue
            b(col) = b(col) - a(col,piv) * b(piv)
 140     continue
c
 200  continue
c
c     *** final check for singularity ***
c
      if (abs(a(n,n)) .lt. tol) then
         errflg = n
      endif
      end
c
c     Backward substitution
c
      subroutine backsb(a,lda,b,x,n)
c
c     *** in/out variables ***
c
      double precision  a(lda,n), b(n), x(n)
      integer           lda, n
c
c     *** local variables ***
c
      integer           row, col
c
      do 330 row = 1, n
         x(row) = b(row)
 330  continue
      do 350 col = n, 1, -1
         x(col) = x(col) / a(col,col)
         do 340 row = 1, col-1
            x(row) = x(row) - a(row,col) * x(col)
 340     continue
 350  continue
```

```
c
      return
      end
```

B Data-parallel Linear Equation Solver, local pivot search variant

The pivoting is done locally on the processor that owns the pivot column; no intermediate results are broadcasted.

```
      subroutine gelim(a,lda,b,n,tol,perm,errflg)
c
c     *** in/out variables ***
c
      double precision a(lda,n), b(n), tol
      integer          lda, n, perm(lda), errflg
c
c     *** automatic arrays ***
c
      double precision abspiv(n)
      integer          ppiv(n)
c
c     *** local variables ***
c
      double precision dtmp, rpiv
      integer          piv, ppiv, row, col, itmp
c
c     *** distribution and alignment ***
c
chpf$ distribute a(*,cyclic)
chpf$ align ppiv(col) with a(*,col)
chpf$ align abspiv(col) with a(*,col)
chpf$ distribute b(*)
chpf$ distribute perm(*)
c
      errflg = 0
      do 200 piv = 1, n-1
c
c        *** pivot search ***
c
         ppiv(piv) = piv
         abspiv(piv) = abs(a(piv,piv))
         do 110 row = piv+1, lda
            if (abs(a(row,piv)) .gt. abspiv(piv)) then
               abspiv(piv) = abs(a(row,piv))
               ppiv(piv)   = row
            endif
 110     continue
c
c        *** distribute pivot ***
c
         ipiv = ppiv(piv)
```

```
         rpiv = abspiv(piv)
c
c        *** check for singularity ***
c
         if (rpiv .lt. tol) then
           errflg = ipiv
           write (*,*) 'tolerance exceeded: ', rpiv
           return
         else if (ipiv .ne. piv) then
c
c           *** permutation in a ***
c
            do 120 col = piv, n
              dtmp        = a(piv,col)
              a(piv,col)  = a(ipiv,col)
              a(ipiv,col) = dtmp
 120        continue
c
c           *** permutation in b ***
c
            dtmp    = b(piv)
            b(piv)  = b(ipiv)
            b(ipiv) = b(piv)
c
c           *** permutation in perm ***
c
            itmp = perm(piv)
            perm(piv) = perm(ipiv)
            perm(ipiv) = itmp
         end if
c
c        *** elimination ***
c
         do 140 row = piv+1, n
            a(row,piv) = a(row,piv) / a(piv,piv)
            b(row) = b(row) - a(row,piv) * b(piv)
            do 130 col = piv+1, lda
               a(row,col) = a(row,col) - a(row,piv) * a(piv,col)
 130        continue
 140     continue
c
 200  continue
c
c     *** final check for singularity ***
c
      if (abs(a(n,n)) .lt. tol) then
        errflg = n
      endif
c
      return
      end
```

C Data-parallel Linear Equation Solver, all local variant

Pivot search, row exchange, and elimination are independent for all (distributed) columns; see Table (cf. 7) for results.

```
      subroutine gelim(a,lda,b,n,tol,perm,pcol,errflg)
c
c     *** in/out variables ***
c
      double precision  a(lda,n), b(n), tol
      integer           lda, n, perm(n), pcol(n), errflg
c
c     For programming reasons pcol should have been
c     an automatic array; however, in that case the XHPF compiler
c     would compute indexes for pcol inside the copy-loop.
c
c     *** local variables ***
c
      integer           row, col, piv, ppiv, itmp, npiv
      double precision  abspiv, dtmp
c
c     *** intrinsic functions ***
c
      intrinsic         abs
      double precision  abs
c
c     *** distribution and alignment ***
c
chpf$ distribute  a(*,cyclic)
chpf$ distribute  b(*)
chpf$ distribute  pcol(*)
chpf$ distribute  perm(*)
c
      do 200 piv = 1, n-1
c
c        *** copy pivot column into pcol (replicated) ***
c
         do 400 row = 1, lda
            pcol(row) = a(row,piv)
 400     continue
c
c        *** pivot search ***
c
         ppiv = piv
         abspiv = abs(pcol(piv))
         do 110 row = piv+1, lda
            if (abs(pcol(row)) .gt. abspiv) then
               abspiv = abs(pcol(row))
               ppiv   = row
            endif
 110     continue
c
c        *** check for singularity ***
c
         if (abspiv .lt. tol) then
```

```
          errflg = ppiv
          write (*,*) 'tolerance exceeded: ', abspiv
          return
        else if (ppiv .ne. piv) then
c
c          *** permutation in a ***
c
           do 120 col = 1, n
              dtmp        = a(piv,col)
              a(piv,col)  = a(ppiv,col)
              a(ppiv,col) = dtmp
 120       continue
c
c          *** permutation in b ***
c
           dtmp    = b(piv)
           b(piv)  = b(ppiv)
           b(ppiv) = dtmp
c
c          *** permutation in perm ***
c
           itmp       = perm(piv)
           perm(piv)  = perm(ipiv)
           perm(ipiv) = itmp
           *** permutation in pcol ***
           dtmp       = pcol(piv)
           pcol(piv)  = pcol(ipiv)
           pcol(ipiv) = dtmp
        end if
c
c       *** elimination ***
c
        do 125 row = piv+1, lda
           pcol(row) = pcol(row) / pcol(piv)
 125    continue
        do 140 col = piv+1, lda
           do 130 row = piv+1, n
              a(row,col) = a(row,col) - pcol(row) * a(piv,col)
 130       continue
 140    continue
        do 150 row = piv+1, lda
           b(row) = b(row) - pcol(row) * b(piv)
 150    continue
c
 200  continue
c
c     *** final check for singularity ***
c
      if (abs(a(n,n)) .lt. tol) then
        errflg = n
      endif
c
      return
      end
```

D Backward substitution

This section describes the column- and row-oriented versions of the backward substitution. Since we implemented the row-oriented version, only this version has the forced optimizations.

```
      subroutine backsb(a,lda,b,x,n)
c
c     ***** column-oriented backward substitution *****
c
c     *** in/out variables ***
c
      integer           lda, n
      double precision  a(lda,n), b(n), x(n)
c
c     *** local variables ***
c
      double precision  dtmp
      integer           row, col
c
c     *****      distribution and alignment       *****
c     *****  in general for this type of routine  *****
c     *****  the distribution is done outside the *****
c     *****              subroutine               *****
c
chpf$ distribute x(cyclic)
chpf$ align a(*,i) with x(i)
chpf$ distribute b(*)
c
      do row = 1, n
         x(row) = b(row)
      end do
c
      do col = n, 1, -1
        x(col) = x(col) / a(col,col)
        do row = col+1, n
          x(row) = x(row) - a(row,col)*x(col)
        end do
      end do
c
      return
      end
```

```
      subroutine backsb(a,lda,b,x,n)
c
c     ***** row-oriented backward substitution *****
c
c     *** in/out variables ***
c
      integer           lda, n
      double precision  a(lda,n), b(n), x(n)
```

```
 9 c
10 c     *** local variables ***
11 c
12       double precision  dtmp
13       integer           row, col
14 c
15 c     *****      distribution and alignment       *****
16 c     *****  in general for this type of routine  *****
17 c     *****  the distribution is done outside the *****
18 c     *****              subroutine               *****
19 c
20 chpf$ distribute x(cyclic)
21 chpf$ align a(*,i) with x(i)
22 chpf$ distribute b(*)
23 c
24       do row = n, 1, -1
25          dtmp = 0.d0
26 capr$    do par on x<1+row~1>
27 capr$    ignore preloop com
28          do col = row+1, n
29             dtmp = dtmp + a(row,col) * x(col)
30          end do
31          x(row) = b(row) - dtmp
32          x(row) = x(row) / a(row,row)
33       end do
34 c
35       return
36       end
```

E Propagation

The interprocedural analysis propagates parallelization problems.

```
 1       subroutine gcro(...)
 2 c
 3       double precision r(n), uu(n,m+1)
 4       double precision x(n), cc(n,m+1)
 5       double precision ri(n), vv(n,m+1)
 6       double precision hh(m+1,m)
 7 c
 8 chpf$ distribute r(block)
 9 chpf$ align uu(i,*) with r(i)
10 chpf$ align cc(i,*) with r(i)
11 chpf$ align vv(i,*) with r(i)
12 c

          ⋮

13       call gmreso(n,m,k,r,uu(1,k+1),ri,...)
14       k = k+1
15       do i = 1, n
16          cc(i,k) = r(i) - ri(i)
17       end do
```

```
18        cn(ksize) = sqrt(ddot(n,cc(1,ksize),1,cc(1,ksize),1))
19        call daxpy(n,-d(ksize),cc(1,ksize),1,r,1)
20        call daxpy(n,d(ksize),uu(1,ksize),1,x,1)
```

⋮

```
21        end
22  c
23        subroutine gmreso(n,m,k,b,x,r,...)
24  c
25        double precision vv(n,m+1)
26  c
27  chpf$ inherit ...
28  chpf$ align vv(i,*) with b(i)
29  c
30  c     ***** Arnoldi *****
31        do while ((j.le.m) .and. (.not.conv))
32           call matvec(n,vv(1,j),vv(1,j+1),...)
33  c        ***** orthogonalize on cc_1 .. cc_k *****
34           do i = 1, k
35              bb(i,j) = ddot(n,cc(1,i),1,vv(1,j+1),1)/(cn(i)**2.d0)
36              call daxpy(n,-bb(i,j),cc(1,i),1,vv(1,j+1),1)
37           end do
38  c        ***** orthogonalize on vv_1 .. vv_j *****
39           do i = 1, j
40              hh(i,j) = ddot(n,vv(1,i),1,vv(1,j+1),1)
41              call daxpy(n,-hh(i,j),vv(1,i),1,vv(1,j+1),1)
42           end do
43        end do
```

⋮

```
44  c     ***** compute inner solution *****
45        do i = 1, j-1
46           call daxpy(n,rsh(i),vv(1,i),1,x,1)
47        end do
```

⋮

```
48  c     ***** compute inner residual *****
49        do i = 1, j
50           call daxpy(n,-xl(i),vv(1,i),1,r,1)
51        end do
52        ***** oblique orthogonalization of new u_k+1 *****
53        do i = 1, k
54           call daxpy(n,-x0(i),uu(1,i),1,x,1)
55        end do
```

⋮

```
56        end
```