

# Distribution Assignment Placement: Effective Optimization of Redistribution Costs

Jens Knoop, *Member, IEEE Computer Society*, and Eduard Mehofer, *Member, IEEE Computer Society*

**Abstract**—Data locality and workload balance are key factors for getting high performance out of data-parallel programs on multiprocessor architectures. Data-parallel languages such as High-Performance Fortran (HPF) thus offer means allowing a programmer both to specify data distributions, as well as to change them dynamically in order to maintain these properties. On the other hand, redistributions can be quite expensive and significantly degrade a program's performance. They must thus be reduced to a minimum. In this article, we present a novel, aggressive approach for avoiding unnecessary remappings which works by eliminating *partially dead* and *partially redundant* distribution changes. Basically, this approach evolves from extending and combining two algorithms for these optimizations achieving each on its own optimal results. In distinction to the sequential setting, the data-parallel setting leads naturally to a family of algorithms of varying power and efficiency allowing requirement-customized solutions. The power and flexibility of the new approach are demonstrated by various examples, which range from typical HPF fragments to real world programs. Performance measurements underline its importance and show its effectivity on different hardware platforms and different settings.

**Index Terms**—Data-parallel languages, High-Performance Fortran (HPF), dynamic data redistribution, data flow analysis, optimization, partially dead and partially redundant assignment elimination.

## 1 INTRODUCTION

A major difficulty in using *High-Performance Computing Systems* (HPCs) is the fact that the programmer is often forced to deal with all aspects of parallelization explicitly. This results in a programming style which can be compared to assembly language programming on sequential machines. It leads to particularly slow software development cycles and high software costs which, in consequence, is a main obstacle for a wide acceptance of HPCs. Data-parallel languages such as *High-Performance Fortran* (HPF) [12], *Fortran D* [14], and *Vienna Fortran* [42], have been proposed to enable users to design programs at a high level in much the same way as they are accustomed to on sequential machines. Sometimes, however, big performance gaps between data-parallel program versions and hand-coded message passing versions may show up. Though HPF compilers are maturing in handling advanced features efficiently and substantial progress has been made in both research and commercial compilers, the wide acceptance of data-parallel languages depends crucially on the compilers' ability of generating consistent highly performant code for a broad range of applications. This still demands new and more powerful compiler optimizations.

**The Goal: Avoiding Unnecessary Remappings.** In this paper, we contribute to this demand by presenting a novel, aggressive approach for reducing redistributions in

advanced, scientific applications.<sup>1</sup> The user-controlled distribution of data across the local memories of processing nodes is a central feature of data-parallel languages. As claimed in [36] (see chapter 6) and shown by (some of) the examples in Section 2, scientific applications typically consist of different computational phases representing different basic algorithms to be applied to the problem data owning each a best data distribution. Dynamic redistributions have proven useful in maintaining data locality and workload balance in these cases. On the other hand, redistributions can be quite expensive and must be reduced to a minimum. Instead of putting this burden on the programmer, the compiler can perform such optimizations automatically, allowing the programmer to concentrate on correctness and transparency of the application. This is particularly important, if not indispensable, for languages such as HPF since not all redistributions are explicitly visible to the programmer, but may be implicitly inserted by the compiler. Consider the program fragment of Fig. 1. The redistribute directive at Line (1) results in an implicit remapping of array B. If array B is not used along program paths from (1) to (2) and dead remappings are not eliminated by the compiler, the programmer has to specify the mapping of array B by means of a distribute directive instead of an alignment directive in order to do dead code elimination manually. At Lines (2) and (3), both alignment and distribution is changed resulting in two remappings of array B whereby the first one is dead assuming that there is no use of B between (2) and (3). Note that this cannot be resolved by permuting Lines (2) and (3)—in that case two

- J. Knoop is with the Department of Computer Science, University of Dortmund, Baroper Straße 301, D-44221 Dortmund, Germany.  
E-mail: knoop@ls5.cs.uni-dortmund.de.
- E. Mehofer is with the Institute of Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria.  
E-mail: mehofer@par.univie.ac.at.

Manuscript received 30 Aug. 1999; accepted 29 Nov. 2001  
For information on obtaining reprints of this article, please send e-mail to: tps@computer.org, and reference IEEECS Log Number 110498.

1. Throughout this paper, we are using the terms *redistribution* and *remapping* synonymously. In HPF, the term *remapping* refers to each reassignment of data elements to processor memories covering in this way redistribute/realign directives and procedure calls.

```

!HPF$ DYNAMIC, DISTRIBUTE(BLOCK,*) :: A
!HPF$ DYNAMIC, ALIGN WITH A :: B
...
(1) !HPF$ REDISTRIBUTE A(*,BLOCK)
...
(2) !HPF$ REALIGN B(i,j) WITH A(j,i)
...
(3) !HPF$ REDISTRIBUTE A(BLOCK,*)
...
(4) CALL FOO(A)
(5) CALL BAR(A)

```

Fig. 1. Well written HPF and remappings.

remappings of array B are performed as well. Again, only the compiler can eliminate such dead remappings. Finally, the subroutine calls at Lines (4) and (5) result in implicit remappings of array A on entry and exit of subroutines FOO and BAR, respectively, in order to establish the distributions requested by the subroutines on entry and to restore the original ones on exit. If subroutines FOO and BAR expect the same distribution for their dummy argument (e.g., (CYCLIC,\*)), restoring the original distribution on exit of subroutine FOO is dead and can be eliminated which turns the remapping on entry of subroutine BAR to be redundant and, hence, can be eliminated thereafter as well. Note that the programmer has no possibility to avoid these remappings and has to rely on the compiler to perform appropriate optimizations.

### The Approach: Distribution Assignment Placement (DAP).

Our approach works by eliminating *partially dead* and *partially redundant* distribution changes. The essence of these notions is illustrated in Figs. 2 and 3 using a classical setting. In Fig. 2a, the assignment to variable  $a$  in node 1 is partially dead as the value of  $a$  is not used along paths following the left branch to node 2. Dually, in Fig. 3a, the assignment to  $a$  in node 3 is partially redundant as value  $b+c$  is already contained by  $a$ , if node 3 is reached along a path via node 1. These notions can be enhanced to distribution changes. A distribution change is partially dead at a program point if it is not used along some program paths leaving this point. Conversely, it is partially redundant there, if there is a path along which the distribution under consideration is already established when reaching it. Note that the notions of partially dead and partially redundant distribution changes subsume those of totally dead and totally redundant distribution changes, i.e., distribution changes, which are not used on any program continuation or have been established on every path reaching the program point under consideration.

Basically, our approach evolves from extending and combining two algorithms for *partially dead code elimination* (PDCE) and *partially redundant assignment elimination* (PRAE) for sequential programs, which achieve each on its own optimal results (cf. [27], [28]). Intuitively, the new algorithm computes beneficial insertion points for redistributions by means of codes *hoisting* and *sinking* in order to make partially redundant and partially dead distribution assignments totally redundant and totally dead, which can

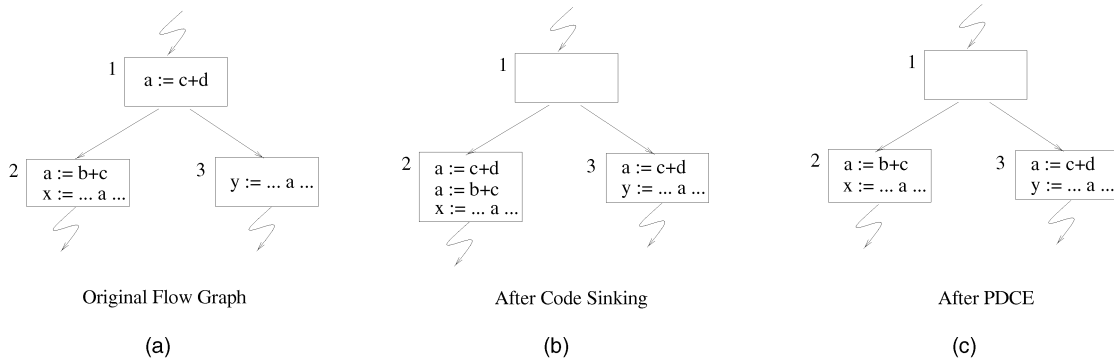


Fig. 2. Illustrating partial deadness and its elimination.

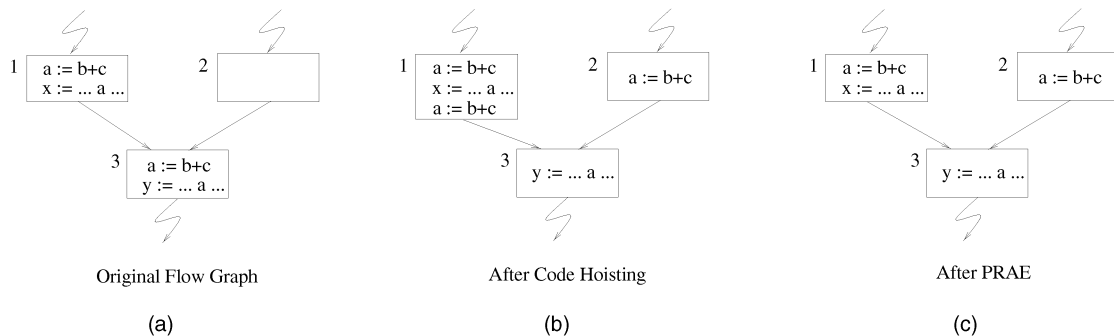


Fig. 3. Illustrating partial redundancy and its elimination.

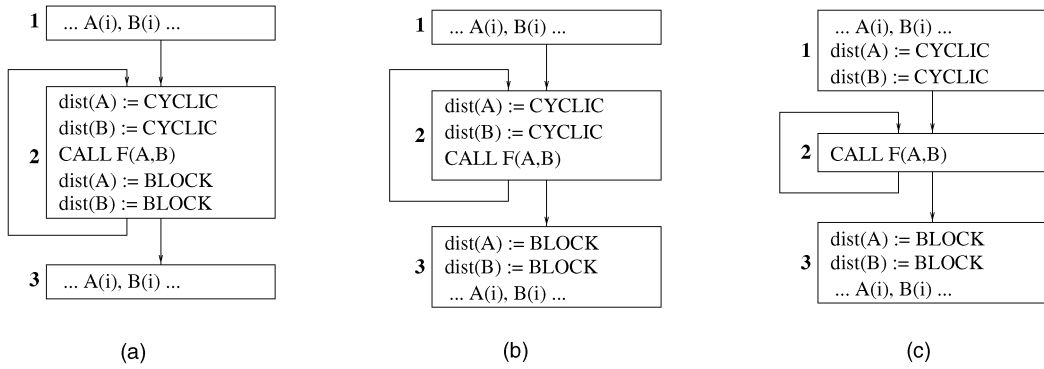


Fig. 4. First example: Illustrating the basic phenomena. (a) Original flow graph, (b) after the PDCE-step, and (c) optimized flow graph.

then be eliminated. This is illustrated in Figs. 2b and 3b. By inserting new occurrences of  $a := c + d$  in nodes 2 and 3 in Fig. 2b, and of  $a := b + c$  in nodes 1 and 2 in Fig. 3b, the previously partially dead assignment of  $a := c + d$  in node 1 of Fig. 2a becomes totally dead and the partially redundant assignment of  $a := b + c$  in node 3 of Fig. 3a becomes totally redundant and can be eliminated as shown in Figs. 2c and 3c.

Interleaving the four elementary transformations of 1) code hoisting and 2) sinking and 3) dead-code and 4) redundant-code elimination, captures uniformly the removal of unnecessary remappings in straight-line code, as well as in loops. Besides the well-known *second-order effects* (cf. Section 2) introduced by interdependences between different statement patterns, which can as usual be overcome by repeated applications of the elementary transformations, interleaving of all four elementary transformations reveals additional interdependences between the transformations themselves: The final result depends on the particular sequence of the elementary transformations, which requires a refined optimality consideration. In contrast to the sequential setting, however, the data-parallel setting leads naturally to a family of algorithms of varying power and efficiency allowing requirement-customized solutions. The basic algorithms of this family are the algorithms for *Pure DAP* and *Full DAP*. While *Pure DAP* focuses on redistributions and resolves second-order effects between them, *Full DAP* resolves second-order effects between all statements, i.e., ordinary assignments and redistributions. Partially dead and partially redundant assignments remaining in the program cannot be eliminated further by means of the elementary transformations without changing its branching structure or impairing some of its executions.<sup>2</sup> Finally, we want to stress that our results, as well as framework, are general and not limited to HPF distribution assignment placement. It can be applied in other contexts as well, whenever partial dead-code elimination and partial redundancy elimination is combined.

**Structure of the Paper.** In Section 2, we motivate our optimization by various examples, which range from

2. While the latter constraint of (at least) performance preservation along all paths is self-evident, the first one of branching structure preservation is typical for program optimization. Dropping it in order to eliminate all partially dead and redundant statements in the fashion, e.g., of the systematic approach of [38], causes in the worst case an exponential increase of the program size and the introduction of nondeterminism into the program.

typical HPF fragments to real world programs, demonstrating thereby the power, flexibility, and relevance of our approach. Subsequently, we discuss the related work in Section 3 and summarize the basic terms required for the formal development of our approach in Section 4. Section 5 is then central, where we stepwise develop our algorithms for *Pure DAP* and *Full DAP*. Subsequently, we construct a variety of practically important variants of these algorithms, which constitute a family of *DAP*-algorithms of varying power and efficiency. In Section 6, we report on performance measurements underlining the importance of *DAP* before drawing our conclusions in Section 7. The Appendix, finally, presents complementary material, which demonstrates the impact of second-order effects between eliminating partially dead and partially redundant assignments.

## 2 ESSENCE AND RELEVANCE OF DAP

In this section, we first illustrate the essence of our approach and the phenomena underlying it in a context as concise as possible by considering two program fragments. Subsequently, we discuss the relevance of our approach for real world applications considering the HPF Applications suite [33], [41] for illustration. In particular, we demonstrate the effectivity of our technique for the partial differential equation solver ADI (Alternating Direction Implicit) and show that inadequate compiler support for distribution management leads to artificial and intransparent programming styles on the user side in order to program around the traps of compiler shortcomings.

Before going into details, however, we briefly recall the notion of *distribution assignments* introduced in [31]: Distribution assignments are generated by the compiler whenever an array is associated with a distribution in the HPF program. This enables a uniform treatment of all sources of distribution changes occurring in an HPF program since redistributions can be achieved in HPF in several ways either with *REDISTRIBUTE/REALIGN* directives or implicitly by procedure calls.

**The Essence of DAP.** The example in Fig. 4 consists of a subroutine call located inside of a loop with statically mapped actual array arguments *A* and *B* and with prescriptively, cyclically mapped dummy arrays. Let us assume that *A* and *B* are mapped noncyclically. Hence, redistributions are performed automatically just before the subroutine call in order to establish the requested distribu-

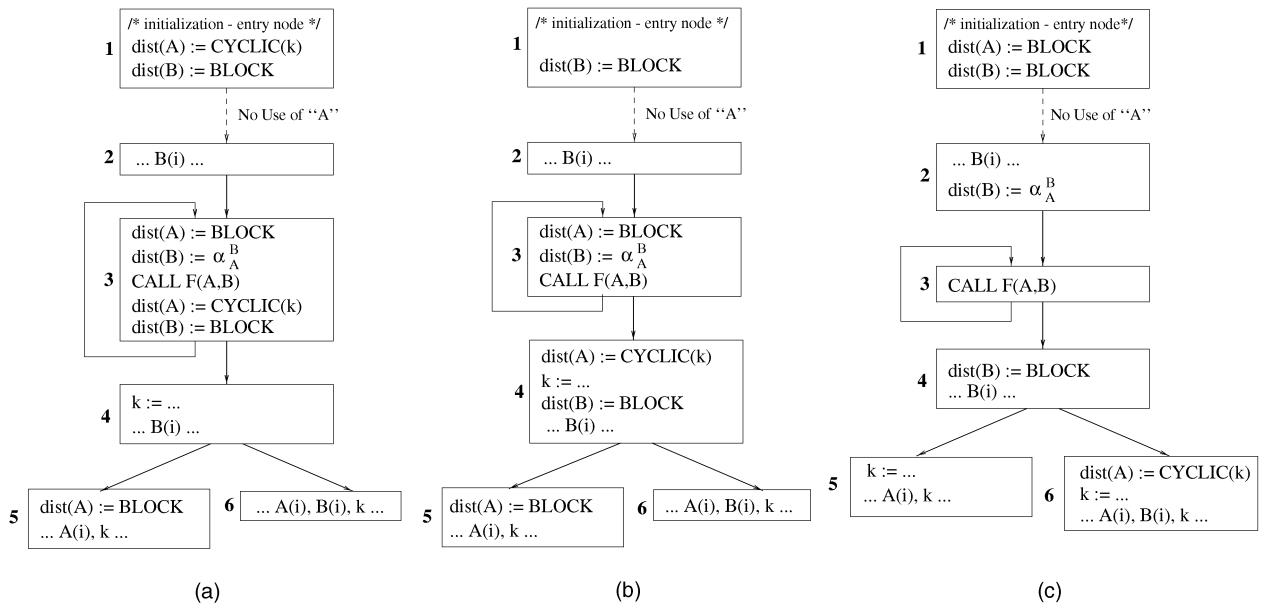


Fig. 5. Second example: Illustrating the impact of second-order effects. (a) Original flow graph, (b) after the PDCE-step, and (c) optimized flow graph.

tions and just after the call to restore the original ones. Note that these implicit remappings are invisible to the programmer. In the example, they are denoted by distribution assignments whereby the right-hand side terms **BLOCK** and **CYCLIC** indicate that the distributions of arrays **A** and **B** are changed to **BLOCK** and **CYCLIC**, respectively. Note that it is a well-written HPF code, which can even be considered quite typical. Think, for example, of calling mathematical library routines from HPF which is of growing importance. In this context, subroutine **F** can be seen as HPF wrapper function which chooses the right distribution and initializes the library descriptors correspondingly. The integration of HPF and highly optimized mathematical libraries is discussed in more detail in Section 6. Returning to the example, it is worth noting that none of the distribution assignments is (totally) redundant or dead. Thus, they cannot be simply eliminated. However, the distribution assignments after leaving subroutine **F** are *partially dead* because they are dead within the loop (but alive with respect to node 3). Thus, they can be removed from the loop as shown in Fig. 4b. As a side-effect, this turns the totally live distribution assignments before entering subroutine **F**, which were originally required in every loop iteration, to be *partially redundant*. They are redundant within the loop (but not redundant with respect to the static distribution), which allows them to be removed from the loop, too, as shown in Fig. 4c. Such effects are usually called *second-order effects* (cf. [37]). Their impact on and their relevance for distribution assignment placement is discussed by means of the example shown in Fig. 5. It illustrates the full power of our approach to resolve second-order effects between different statement patterns.

The use of alignments for specifying the distribution of arrays introduces dependences between distribution assignments. In addition, parameters in distribution specifications such as in **BLOCK**(*i*) or **CYCLIC**(*i*) introduce dependencies between ordinary and distribution assignments as well. Both kinds of dependencies can be the source of second-order effects as illustrated in Figs. 5a, 5b, and 5c. As before,

a subroutine which is transparent for the distributions of arrays **A** and **B** is called within a loop. Array **B** is aligned to array **A** on entry of subroutine **F**, denoted by  $\alpha_A^B$ . Analogously to Fig. 4, the distribution assignments at the exit of subroutine **F** can be removed from the loop by placing them in node 4 as they are dead inside the loop (cf. Fig. 5b). Whereas the distribution assignment  $\text{dist}(\mathbf{B}) := \mathbf{BLOCK}$  at node 4 is now totally live, the statement  $\text{dist}(\mathbf{A}) := \mathbf{CYCLIC}(k)$  is still partially dead. Moving it to node 6, where it would be totally live as well, is prevented by the assignment to variable *k* (cf. Fig. 5b). However, moving the ordinary assignment to *k* first suspends this blockade and subsequently allows us to eliminate the partially dead assignment  $\text{dist}(\mathbf{A}) := \mathbf{CYCLIC}(k)$  as desired. As a side-effect, this turns the distribution assignment at node 5 to be totally redundant, which can now be eliminated as well. As another second-order effect, the distribution assignment to **A** at the entry of the loop becomes partially redundant and can be hoisted out of the loop to node 1. This suspends the blockade of the alignment  $\text{dist}(\mathbf{B}) := \alpha_A^B$  at the entry of the loop too, which, as another second-order effect, can now be moved to node 2.

Fig. 5c shows the result of the complete transformation achieved by our approach. Note that the optimized flow graph is optimal: It is free of any partially dead and partially redundant distribution assignment.

**The Relevance of DAP for HPF.** It is well known that data distribution has a crucial impact on runtime performance. The change of a distribution for some poorly performing program phase can be very beneficial [36]. HPF supports redistributions in two ways. In the HPF base language, the procedure call mechanism can be utilized. If the mapping of the actual argument differs from the mapping specified for the dummy argument, an implicit redistribution takes place. The other possibility is based on dynamically mapped arrays which are part of the Approved Extensions of HPF [12]. The distribution of dynamically

<pre> REAL, DIMENSION (N,N) :: x,a,b ... <i>x,a,b are assumed to have some actual mapping</i> DO iter=1,maxiter   <i>l<sub>1</sub> :remap2row: x,a,b</i>   CALL rows(x,a,b)   <i>l<sub>2</sub> :remap2actual: x,a,b</i>   <i>l<sub>3</sub> :remap2column: x,a,b</i>   CALL columns(x,a,b)   <i>l<sub>4</sub> :remap2actual: x,a,b</i> END DO </pre>	<pre> REAL, DIMENSION (N,N) :: x,a,b ... <i>x,a,b are assumed to have some actual mapping</i> DO iter=1,maxiter   <i>l<sub>1</sub> :remap2row: x,a,b</i>   CALL rows(x,a,b)   <i>l<sub>3</sub> :remap2column: x,a,b</i>   CALL columns(x,a,b) END DO <i>l<sub>4</sub> :map2actual: x,a,b</i> </pre>
(a)	
<pre> SUBROUTINE rows(x,a,b) REAL, DIMENSION (:,:) :: x,a,b !HPF\$ DISTRIBUTE (BLOCK,*) :: x,a,b ! forward and backward sweeps along rows DO J=2,N   DO I=1,N     X(I,J)=X(I,J)-X(I,J-1)*A(I,J)/B(I,J-1)     B(I,J)=B(I,J)-A(I,J)*A(I,J)/B(I,J-1)   END DO END DO DO I=1,N   X(I,N)=X(I,N)/B(I,N) END DO DO J=N-1,1,-1   DO I=1,N     X(I,J)=(X(I,J)-A(I,J+1)*X(I,J+1))/B(I,J)   END DO END DO END </pre>	<pre> SUBROUTINE columns(x,a,b) REAL, DIMENSION (:,:) :: x,a,b !HPF\$ DISTRIBUTE (*,BLOCK) :: x,a,b ! forward and backward sweeps along columns DO J=1,N   DO I=2,N     X(I,J)=X(I,J)-X(I-1,J)*A(I,J)/B(I-1,J)     B(I,J)=B(I,J)-A(I,J)*A(I,J)/B(I-1,J)   END DO END DO DO J=1,N   X(N,J)=X(N,J)/B(N,J) END DO DO J=1,N   DO I=N-1,1,-1     X(I,J)=(X(I,J)-A(I+1,J)*X(I+1,J))/B(I,J)   END DO END DO END </pre>
(b)	

Fig. 6. ADI integration kernel. (a) Unoptimized and optimized version of ADI main loop. (b) Subroutines performing the sweeps.

mapped arrays can be changed explicitly with the executable directives `REALIGN` and `REDISTRIBUTE`. Since sophisticated analysis is required to realize dynamically mapped arrays efficiently, many HPF compilers do not support this feature appropriately.

Hence, even if it would be the most natural way to express the semantics by dynamically mapped arrays, programmers tend to use other programming constructs instead (often at the price of losing transparency). The most rudimentary escape is to copy the data from one array to another explicitly (doubling in this way the memory requirement). This is done, for instance, in the Wavelet Image Processing code of the HPF Applications suite [33], [41]. Array remappings can also be achieved, albeit in a very limited manner, by using the Fortran 90 intrinsic function `TRANPOSE`, see, for example, Solution of two-dimensional Poisson Equation, two-dimensional Fast Fourier Transformation, or two-dimensional Convolution of the HPF Applications suite. The growing relevance of DAP in the context of combining HPF and highly optimized mathematical libraries is discussed in more detail in Section 6.

The most natural way to express array remappings without using dynamically mapped arrays is to employ the HPF procedure call mechanism. Consider the ADI (Alternating Direction Implicit) integration kernel [30], a well-known partial differential equation solver. The kernel

consists of forward and backward sweeps along the rows followed by forward and backward sweeps along the columns. Hence, a row-wise data distribution requires communication for the sweeps along the columns but none along the rows—a column-wise data distribution requires communication vice versa. Choosing a row-wise distribution for the sweeps along the rows and a column-wise distribution for the column sweeps and redistributing the arrays between the two phases, no communication and synchronization is required within the sweeps at the expense of the remapping overhead. Fig. 6 shows the unoptimized and optimized program version of ADI taken from [30], where, however, instead of dynamically mapped arrays, subroutines are used to perform the sweeps along the rows and columns, respectively. This reflects the current status of HPF-2 [12], where dynamically mapped arrays have been moved from the base HPF language to the Approved Extensions.

As shown in Fig. 6, the unoptimized version of the ADI main loop performs four remappings of the arrays  $x$ ,  $a$ , and  $b$  just before and just after the subroutine calls `rows` and `columns` in order to establish the distributions requested by the dummy arguments and to restore the original ones, respectively. Partial dead-code elimination recognizes the remapping at  $l_2$  as dead. The remapping at  $l_4$  is identified as

dead within the loop but live with respect to the code following the loop. This partial deadness can be eliminated by moving it just after the loop.

Summarizing, the 12 remappings required within one iteration for the arrays  $x$ ,  $a$ , and  $b$  in the unoptimized version are reduced to six remappings within the loop and three remappings after the loop in the optimized program version. The number of remappings which are performed in the optimized ADI version less than in the unoptimized version is given by  $r_k$  with  $k$  denoting the executed iterations as follows:  $r_1 = 3$ ,  $r_2 = 9$ ,  $r_5 = 27$ ,  $r_{10} = 57$ ,  $r_{50} = 297$ , and  $r_{100} = 597$ . Performance results showing the (dramatic) effect of this optimization are given in Section 6.

Note that, in the example in Fig. 6, there are no partially redundant distribution changes. Hence, related approaches based on partial redundancy elimination fail on this example (cf. Section 3). In fact, reducing redistributions to a minimum requires generally the combined effects of partial dead-code elimination and partial redundancy elimination.

### 3 RELATED WORK

Approaches aiming at reducing distribution costs by avoiding unnecessary redistributions have been proposed by several authors, often together with empirical evidence showing the importance of such approaches for the goal of obtaining highly efficient code.

One of the first approaches has been proposed by Hall et al. [18]. They showed that a straightforward placement of mapping routines produces usually highly inefficient code, which strongly demands for optimizations. Based on reaching mapping and live mapping analysis, they proposed an interprocedural algorithm capable of removing totally redundant and dead remappings. In addition, their algorithm moves loop-invariant remappings after the loop or prior the loop. However, it does not consider the more general problem of eliminating partially dead and partially redundant remappings. Applied to the example in Fig. 5a, the remappings inside the loop are thus only moved just after and just before the loop, i.e., to the nodes 4 and 2. Hence, partial deadness, however, introduced by if-statements, as in Fig. 5b, for the remapping  $\text{dist}(\mathbf{A}) := \text{CYCLIC}(k)$  are not eliminated.

A different approach has been proposed by Coelho and Ancourt. In [7], they describe compilation techniques which optimize remappings by removing useless ones and taking advantage of replications to shorten individual remappings. Optimality in their sense means that for a given remapping, a minimal number of messages is sent over the network. Elimination of useless remappings is based on a new representation called remapping graph which is a subgraph of the control flow graph. The nodes of the graph represent remappings and edges denote possible paths in the control flow graph with the same array being remapped at both sides. Code motion is not applied in order to reduce the number of executed remappings further.

Like Hall et al., Palermo et al. [34] presented an interprocedural approach. It comprises several analyses which are important to compile dynamically mapped arrays efficiently. Their algorithm determines first the

distributions which are present at every program point by reaching distribution analysis. Based on this information, it optimizes the transition between dynamic distributions caused by redistributions. The optimizations include the elimination of useless remappings and motion of loop-invariant redistributions. However, "iterated" second-order effects as in the example in Fig. 15 of Appendix B illustrating the power of our approach, are not captured by this approach.

In addition to the approaches above, several authors proposed communication optimization frameworks relying on code motion techniques for *partial redundancy elimination* (PRE). For example, Gupta et al. [17] applied PRE-techniques to available section descriptors and perform a number of optimizations in order to reduce the communication in a program. Agrawal et al. [1] focused on interprocedural placement of communication statements. Both approaches are based on the bidirectional PRE-framework of Dhamdhere and Patil [8]. In contrast, the constraint-based communication placement framework proposed by Kennedy and Sethi [21] is based on the unidirectional PRE-algorithm for *lazy code motion* developed by Knoop et al. [25]. The communication optimization approach presented by Fahringer and Mehofer [11] uses code motion as well for trying to find a good compromise between enhancing communication latency hiding and reducing the number of messages.

As demonstrated above, however, pure PRE-techniques are insufficient in order to reduce redistributions to a minimum by strategic placement of distribution assignments. In general, this requires the combined effects of eliminating *partially dead* and *partially redundant* instructions [24]. In [27], [28], these problems have been separately investigated for a sequential setting. In this article, we show how to enhance and combine these algorithms in order to arrive at a family of DAP-algorithms for a data-parallel setting.

### 4 PRELIMINARIES

As is common in optimization, we represent programs by *directed flow graphs*  $G = (N, E, s, e)$  with node set  $N$  and edge set  $E$ . Nodes  $n \in N$  represent *basic blocks* of instructions  $\iota$ , edges  $(m, n) \in E$  the nondeterministic branching structure of  $G$ , and  $s$  and  $e$  its unique *start node* and *end node*, which are assumed to be free of incoming and outgoing edges, respectively. Additionally, we denote the set of all immediate successors and predecessors of a node  $n$  by  $\text{succ}(n) =_{df} \{m \mid (n, m) \in E\}$  and  $\text{pred}(n) =_{df} \{m \mid (m, n) \in E\}$  and assume that all nodes of  $G$  lie on a path from  $s$  to  $e$ . Moreover, all statements occurring in a program are classified as follows: (ordinary) *assignment statements* involving both scalar and indexed variables; the *empty statement skip*; *distribution assignments* of the form  $\text{dist}(\mathbf{A}) := \delta$  with  $\mathbf{A}$  denoting an array and  $\delta$  an arbitrary distribution, which are generated by the compiler and uniformly express distribution changes occurring throughout the program at subprogram boundaries and at program points with programmer given redistribute/realign directives: *subprogram calls* and *output operations* of the form  $\text{out}(t)$  forcing each operand of term  $t$  to be alive.<sup>3</sup> Finally, all edges starting in a node with

more than one successor and leading to a node with more than one predecessor are assumed to be split by a synthetic node (cf. [22]). This is typical for code motion transformations in order to avoid that the motion process is blocked (cf. [25], [26]).

## 5 DISTRIBUTION ASSIGNMENT PLACEMENT

In this section, we stepwise develop our family of DAP-algorithms rendering possible the choice of requirement-customized solutions for the elimination of unnecessary overhead due to distribution changes. In essence, the algorithms differ from each other in how they exploit the trade-off between efficiency and power of the transformation. The kernel of this family is constituted by the algorithms of *Pure DAP* and *Full DAP*, which we present first. For convenience, we consider an arbitrary but fixed flow graph  $G$ , which allows us an unparameterized notation. Moreover, we denote the set of ordinary and distribution assignment patterns  $\alpha$ , i.e., string symbols of the form  $\alpha \equiv lhs := rhs$  or  $\alpha \equiv \text{dist}(A) := \delta$ , having occurrences in  $G$  by  $\mathcal{P}$ . More specifically, we denote its subset of distribution patterns by  $\mathcal{D}$ ,  $\mathcal{D} \subseteq \mathcal{P}$ .

As illustrated in Section 2, the essence of DAP is to avoid unnecessary executions of distribution assignments at runtime. Intuitively, a distribution assignment is *unnecessary*, either if it is *dead*, i.e., there is no program continuation on which its left-hand side variable is used without a preceding distribution assignment; or, if it is *redundant*, i.e., on every program path reaching it, a distribution assignment of the same pattern has been executed without an intermediate distribution change. More generally, DAP relies on the combined effects of eliminating 1) *partially dead* and 2) *partially redundant* assignments. This is important since both subproblems can optimally be solved as demonstrated in [27], [28], where algorithms for *partially dead code elimination* (PDCE) and *partially redundant assignment elimination* (PRAE) have been proposed, respectively. In this section, we will show how to enhance these algorithms, which have originally been designed for a standard sequential program setting, to the data-parallel setting and how to combine them uniformly in order to arrive at the basis of a family of requirement-customized DAP-algorithms.

### 5.1 Partially Dead Code Elimination

PDCE consists conceptually such as DAP of two elementary transformations: *assignment sinkings* (AS) and *dead code eliminations* (DCE). Assignment sinkings move assignments as far as possible in the direction of the control flow (i.e., while maintaining the program semantics). Intuitively, this places them in a context as *specific* as possible and thereby maximizes the potential of dead code, which subsequently is eliminated by a dead variable analysis. Table 1, where  $N$  and  $X$  are used as abbreviations of “eNtry” and “eXit,” summarizes the data flow analyses for assignment sinkings and dead assignment eliminations, which uniformly capture

ordinary and distribution assignments. As usual, these analyses are specified in terms of an equation system each, whose greatest solutions characterize the set of program points enjoying the property under consideration. The greatest solutions, denoted by  $N - \text{SINK}^*$ ,  $X - \text{SINK}^*$ , and  $X - \text{DEAD}^*$  in Table 1 can be computed by a standard fixed point algorithm (cf. [19], [32]).<sup>4</sup> Subsequently, a new copy of the statement under consideration is inserted at the entry and exit of each node, satisfying the predicate  $N - \text{SINK}^*$  and  $X - \text{SINK}^*$ , respectively, while each instruction located at a node satisfying the predicate  $X - \text{DEAD}^*$  is removed. Note that  $=_{df}$  means “equal by definition.” Hence, computing the insertion and elimination points does not require the solution of an equation system. In the following, we discuss the intuition underlying both analyses.

Basically, the AS-analysis is controlled by the local predicates **LocBlock** and **LocSink**. These are the handles for restricting assignment sinkings to *admissible* ones, i.e., semantics preserving ones. In essence, admissibility requires that assignments are never moved across instructions *blocking* them (expressed by **LocBlock**), i.e., using or modifying their left-hand side variables, or modifying some of their right-hand side variables complemented by variables used in index expressions of their left-hand side variables. Additionally, subprogram calls are considered blockades, too, as we do not perform an interprocedural analysis. In essence, the same blocking constraints we impose on distribution assignments: Distribution assignments of the form  $\text{dist}(A) := \delta$  are blocked by procedure calls and by any occurrence of  $A$  and any definition of a variable used in  $\delta$ . In particular, this guarantees that assignment sinkings respect the distribution proposed by the programmer: For each array reference, the distributions in the original and the optimized program are identical. Note that this constraint could be weakened according to the quite typical assumption for data-parallel languages that array distributions do not affect the program semantics. In our approach, this can easily be achieved by defining the predicate **LocBlock** for ordinary and distribution assignments differently. The second predicate, **LocSink**, identifies all assignment occurrences that can be moved at all: *sinking candidates* are occurrences of ordinary as well as distribution assignments inside a basic block which are *downwards exposed*, i.e., not blocked by a subsequent assignment of the same basic block.

The elimination of dead assignments is based on a (standard) dead variable analysis (cf. [19], [32]). It is controlled by the local predicates **Used** and **Mod**. An occurrence of an ordinary assignment with left-hand side variable  $x$  is dead if  $x$  is dead, i.e., on every program continuation every right-hand side occurrence of  $x$  complemented by occurrences of  $x$  in left-hand side index expressions (expressed by **Used**) is preceded by a modification of  $x$  (expressed by **Mod**). For distribution assignments, the local predicates have to be adapted accordingly. The distribution of an array  $A$  is accessed by every ordinary assignment with left-hand side  $A$  and every

3. In practice also, conditions in if-statements and assignments to global variables (variable whose declaration is outside the scope of the flow graph under consideration) must be treated analogously. This is straightforward and, thus, we omit details here, which can be found in [22].

4. In practice, this can be done simultaneously for all statement patterns and variables using bitvectors (cf. [19], [32]).

TABLE 1  
Assignment Sinking and Dead Variable Analysis

Assignment Sinking Analysis (AS): (Let $\alpha \in \mathcal{P}$ )	
• <b>LocSink<sub>n</sub></b> ( $\alpha$ ):	There is a sinking candidate of $\alpha$ in $n$ .
• <b>LocBlock<sub>n</sub></b> ( $\alpha$ ):	The sinking of $\alpha$ is blocked by some instruction of $n$ .
<b>N-SINK<sub>n</sub></b>	$= \begin{cases} false & \text{if } n = \mathbf{s} \\ \prod_{m \in pred(n)} \mathbf{X-SINK}_m & \text{otherwise} \end{cases}$
<b>X-SINK<sub>n</sub></b>	$= \mathbf{LocSink}_n + \mathbf{N-SINK}_n * \overline{\mathbf{LocBlock}_n}$
Insertion Points: <sup>a b</sup>	
<b>N-INSERT<sub>n</sub></b>	$=_{df} \mathbf{N-SINK}_n^* * \mathbf{LocBlock}_n$
<b>X-INSERT<sub>n</sub></b>	$=_{df} \mathbf{X-SINK}_n^* * \begin{cases} true & \text{if } n = \mathbf{e} \\ \sum_{m \in succ(n)} \overline{\mathbf{N-SINK}_m^*} & \text{otherwise} \end{cases}$
Dead Variable Analysis ( $\equiv$ Dead Code Elimination DCE): <sup>c</sup>	
• <b>Used<sub><math>\iota</math></sub></b> ( $x$ ):	There is a use of the value/distribution of $x$ in $\iota$ .
• <b>Mod<sub><math>\iota</math></sub></b> ( $x$ ):	The left-hand side of the instruction $\iota$ changes the value/distribution of $x$ .
<b>N-DEAD<sub><math>\iota</math></sub></b>	$= \neg \mathbf{Used}_\iota * (\mathbf{X-DEAD}_\iota + \mathbf{Mod}_\iota)$
<b>X-DEAD<sub><math>\iota</math></sub></b>	$= \prod_{i \in succ(\iota)} \mathbf{N-DEAD}_i$
Elimination Points: <sup>a</sup>	$\mathbf{ELIM}_\iota =_{df} \mathbf{X-DEAD}_\iota^*$

- a.  $\mathbf{N-SINK}^*$ ,  $\mathbf{X-SINK}^*$ , and  $\mathbf{X-DEAD}^*$  denote the greatest solutions of the equation systems for sinkability and deadness, respectively.  
b. This means to replace all sinking candidates of the assignment pattern under consideration by instances of this pattern inserted at program points satisfying  $\mathbf{N-INSERT}$  or  $\mathbf{X-INSERT}$ .  
c. For the ease of presentation, equations are specified at instruction level.

right-hand side occurrence of  $A$  complemented by occurrences of  $A$  in lefthand side index expressions. Modifications of distributions are accomplished by distribution assignments.

**The PDCE-Algorithm and its Optimality.** Following [27], the second-order effects induced by interdependences of different assignment patterns (cf. Section 2) are fully captured by repeatedly applying the elementary transformations of assignment sinking and dead code elimination of PDCE to all assignment patterns occurring in the program until it stabilizes. Thus, the complete algorithm is conveniently expressed by means of the following regular-expression like term

$$\text{PDCE} \equiv (\text{AS} + \text{DCE})^+,$$

where  $\text{AS}$  and  $\text{DCE}$  denote a single application of the assignment sinking and dead assignment elimination procedure to all assignment patterns of  $\mathcal{P}$ . As proven in [15], the program finally resulting from PDCE is independent of the specific order of the elementary transformations: It is uniquely determined up to (irrelevant) local reorderings in basic blocks. Moreover, it is *best (optimal)* in the sense that it is *better* than any other program in the set of programs

$$\mathcal{G}_{pdce} =_{df} \{G' \mid G \vdash_{(\text{AS}+\text{DCE})}^* G'\},$$

which denotes the set of all programs, which are derivable from  $G$  by means of sequences of admissible assignment sinkings and dead code eliminations. The precise meaning

of “better” is central here: A program  $G' \in \mathcal{G}_{pdce}$  is *better* than a program  $G'' \in \mathcal{G}_{pdce}$  if and only if, for every assignment pattern, the number of assignments executed on every path in  $G'$  is less or equal to that in  $G''$  (cf. [15], [27]). This is important because it directly implies that the program resulting from PDCE is best whatever the actual execution costs of the occurrences of a specific assignment pattern may be. As we are going to show below, for DAP it is necessary to take execution costs of assignment patterns into account, too.

## 5.2 Partially Redundant Assignment Elimination

PRAE is conceptually dual to PDCE: *assignment hoistings* (AH) move assignments as far as possible in the opposite direction of the control flow. This places them in a context as *universal* as possible, which intuitively maximizes the potential of redundant code that is subsequently eliminated by a *redundant assignment elimination* (RAE). The analyses for AH and RAE are shown in Table 2. The local predicate **LocBlock** for assignment hoisting is defined like its counterpart for assignment sinking. Similarly, this also holds for the local predicate **LocHoist** identifying *hoisting candidates*: These are occurrences of ordinary and distribution assignments inside a basic block which are *upwards exposed*, i.e., not blocked by a preceding instruction of the same basic block.

**The PRAE-Algorithm and its Optimality.** Like for PDCE, repeated applications of assignment hoistings and redundant assignment eliminations suffice to completely capture the second-order effects between different state-

TABLE 2  
Assignment Hoisting and Redundant Assignment Analysis

Assignment Hoisting Analysis (AH): (Let $\alpha \in \mathcal{P}$ )	
• <b>LocHoist</b> <sub><math>n</math></sub> ( $\alpha$ ):	There is a hoisting candidate of $\alpha$ in $n$ .
• <b>LocBlock</b> <sub><math>n</math></sub> ( $\alpha$ ):	The hoisting of $\alpha$ is blocked by some instruction of $n$ .
<b>N-HOIST</b> <sub><math>n</math></sub>	$= \text{LocHoist}_n + \text{X-HOIST}_n * \overline{\text{LocBlock}_n}$
<b>X-HOIST</b> <sub><math>n</math></sub>	$= \begin{cases} false & \text{if } n = e \\ \prod_{m \in \text{succ}(n)} \text{N-HOIST}_m & \text{otherwise} \end{cases}$
Insertion Points: <sup>c d</sup>	
<b>N-INSERT</b> <sub><math>n</math></sub>	$=_{df} \text{N-HOIST}_n^* * \begin{cases} true & \text{if } n = s \\ \sum_{m \in \text{pred}(n)} \overline{\text{X-HOIST}_m^*} & \text{otherwise} \end{cases}$
<b>X-INSERT</b> <sub><math>n</math></sub>	$=_{df} \text{X-HOIST}_n^* * \text{LocBlock}_n$
Redundant Assignment Analysis (RAE): <sup>e</sup> (Let $\alpha \in \mathcal{P}$ such that the left-hand side of $\alpha$ does not occur in its right-hand side term. Let $s$ denote the first instruction of $s$ .)	
• <b>IsOcc</b> <sub><math>\iota</math></sub> ( $\alpha$ ):	Instruction $\iota$ is an assignment of pattern $\alpha$ .
• <b>AssTransp</b> <sub><math>\iota</math></sub> ( $\alpha$ ):	Instruction $\iota$ is transparent for $\alpha$ , i.e., neither $\alpha$ 's left-hand side nor any of the arguments needed in $\alpha$ are modified by $\iota$ (note that $\iota \equiv A := \dots$ is transparent for $\alpha \equiv \text{dist}(A) := \dots$ ).
<b>N-RED</b> <sub><math>\iota</math></sub>	$= \begin{cases} false & \text{if } \iota = s \\ \prod_{i \in \text{pred}(\iota)} \text{X-RED}_i & \text{otherwise} \end{cases}$
<b>X-RED</b> <sub><math>\iota</math></sub>	$= \text{IsOcc}_\iota + \text{AssTransp}_\iota * \text{N-RED}_\iota$
Elimination Points: <sup>c</sup>	<b>ELIM</b> <sub><math>\iota</math></sub> $=_{df} \text{N-RED}_\iota^*$

- c. **N-HOIST**<sup>\*</sup>, **X-HOIST**<sup>\*</sup>, and **N-RED**<sup>\*</sup> denote the greatest solution of the equation systems for hoistability and redundancy, respectively.  
d. This means to replace all hoisting candidates of the assignment pattern under consideration by instances of this pattern inserted at program points satisfying **N-INSERT** or **X-INSERT**.  
e. The analysis is employed at the instruction level, which eases presentation. It can straightforward be modified to work on basic blocks instead.

ment patterns for PRAE. Using the notational convention of the previous section, the complete algorithm is specified by the regular-expression like term

$$\text{PRAE} \equiv (\text{AH} + \text{RAE})^+,$$

where AH and RAE denote a single application of the assignment hoisting and redundant assignment elimination procedure to all assignment patterns of  $\mathcal{P}$ . The program finally resulting from PRAE is uniquely determined up to local reorderings in basic blocks. Moreover, it is *best* (optimal) in the set of programs

$$\mathcal{G}_{prae} =_{df} \{G' \mid G \vdash_{(\text{AH}+\text{RAE})}^* G'\},$$

which denotes the set of programs, which are derivable from  $G$  by sequences of admissible assignment hoistings and redundant assignment eliminations with respect to the relation “better” as defined in Section 5.1 (cf. [28]).

### 5.3 Interdependences between PDCE and PRAE

As recalled above, the elementary transformations of PDCE and PRAE can be applied in any order without affecting the program finally resulting (cf. [27], [28]). In fact, in the set of programs derivable by admissible assignment sinkings (hoistings) and dead (redundant) assignment eliminations, there is a “globally best” one, which is finally reached by every transformation sequence. Unfortunately, this prop-

erty gets lost as soon as all four elementary transformations are interleaved as required for DAP. This is illustrated by the example in Fig. 7. Applying PDCE to the program in Fig. 7a, we first arrive at the program in Fig. 7b; however, applying PRAE, first we arrive at the program in Fig. 7c. Note that both programs are invariant under further admissible assignment motions and dead (redundant) assignment eliminations. Moreover, they are incomparable. While each path through the program fragment of Fig. 7c contains precisely one distribution assignment, there is a path through the fragment in Fig. 7b being free of distribution assignments and another one containing two.

As pointed out by this example, PDCE and PRAE influence each other by mutually removing opportunities for their respective counterpart. In fact, we conjecture that, such as in this example, “maximally transformed” programs are either identical (up to irrelevant reorderings in basic blocks) or incomparable with respect to the order of Section 5.1 counting assignments for every assignment pattern separately. But programs may be considered incomparable for which an order according to another intuitive notion of “better,” which takes individual execution costs of assignment patterns into account, exists. This is demonstrated in the example in Fig. 8. The programs in Figs. 8b and 8c result from the program in Fig. 8a by beginning with PRAE and PDCE, respectively. On paths starting from node 1 the program in Fig. 8c

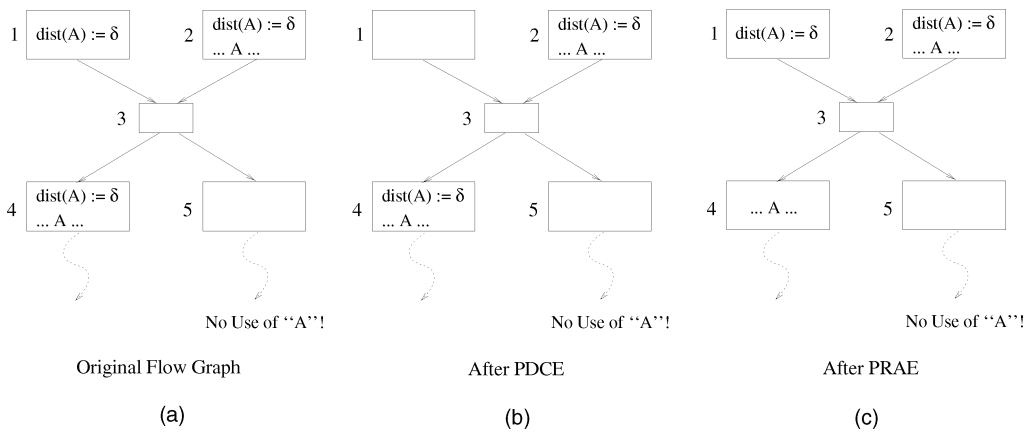


Fig. 7. Interdependences between PDCE and PRAE.

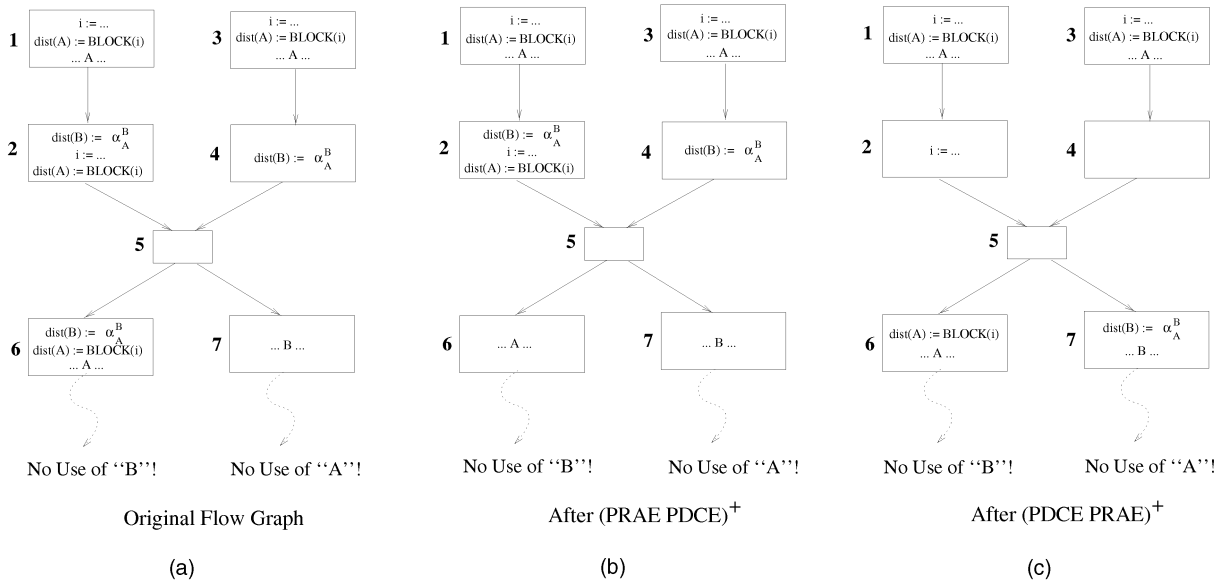


Fig. 8. “Maximally” transformed programs are sometimes comparable.

contains less remappings than the program in Fig. 8b. On the path (3, 4, 5, 7) the same remappings are required for both versions. On the path (3, 4, 5, 6), however, the distribution assignments  $\text{dist}(A) := \text{BLOCK}(i)$  and  $\text{dist}(B) := \alpha_A^B$  are executed in Fig. 8b, whereas in Fig. 8c, the assignment  $\text{dist}(A) := \text{BLOCK}(i)$  is performed twice. Hence, both programs are incomparable with respect to the relation “better” of Section 5.1. However, taking execution costs into account as indicated above, both programs would perfectly be comparable if the execution costs of  $\text{dist}(A) := \text{BLOCK}(i)$  and  $\text{dist}(B) := \alpha_A^B$  coincide. Then, the program in Fig. 8c is strictly “better” than that in Fig. 8b; a fact which is not reflected by the order of Section 5.1.

In practice, validating the equality of costs of different distribution assignment patterns is in general not trivial and requires performance prediction evaluating program paths statically. Masking offers here a way to further improvement which does not rely on such a preprocess. Applied to the program in Fig. 8b, it is important that both remappings on the path from node 3 to node 6 have to be done in any case, while on the corresponding path of the program in Fig. 8c, the second remapping is redundant. By means of

*distribution assignment masking* (cf. Section 5.6), this unnecessary remapping can be suppressed at the price of a much cheaper runtime test, implying that the program in Fig. 8c is “better” than the program in Fig. 8b without having to refer to the relative costs of  $\text{dist}(A) := \text{BLOCK}(i)$  and  $\text{dist}(B) := \alpha_A^B$ . Note, however, that this notion of “better” relies on a specific code generation strategy.

In spite of these subtleties, we have that the process of interleaving all four elementary transformations of PDCE and PRAE always comes up with a program, which cannot be improved any further by means of semantics preserving eliminations of partially redundant and partially dead assignments leaving the program structure invariant, i.e., with a program being “locally best.” In the following, we thus concentrate on the question in which order to apply the elementary transformations in order to achieve fast stabilization. Obviously, AS- and AH-steps should always be interleaved with DCE- and RAE-steps (as AS and AH essentially reverse each other’s effect). This implies interleaving PDCE and PRAE, which, in general, must be applied repeatedly in order to reach a stable state (see Appendix B for an extensive example). In essence, this is a

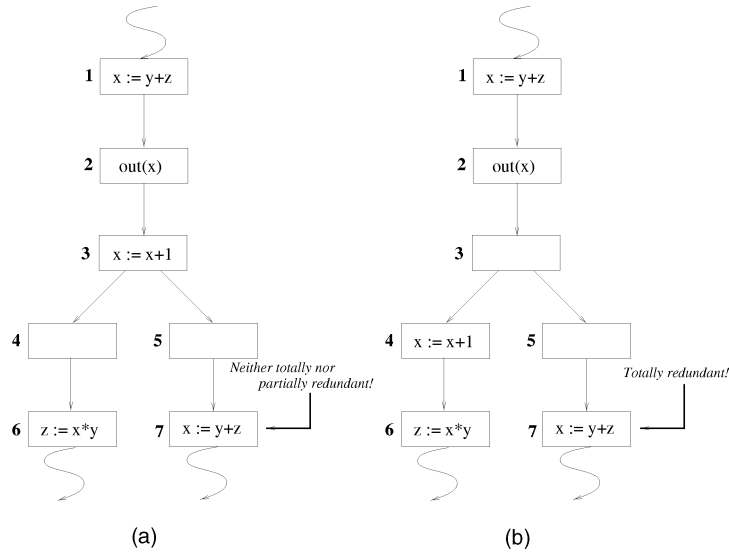


Fig. 9. PDCE may create (partially) redundant assignments.

consequence of the fact that RAE (DCE) removes blockades which prevent partially dead (partially redundant) code to be sunk (hoisted) to places where it becomes totally dead (redundant). Thus, the question reduces essentially to that of starting with PDCE or PRAE. The following facts suggest that starting with PDCE is superior in general. First, in the situation displayed in the example in Fig. 2, which can be considered quite typical for data-parallel programs, PDCE is a prerequisite for enabling PRAE. Second, there is an important difference between PDCE and PRAE concerning their interplay which is summarized in Lemma 1.

**Lemma 1 (PDCE-PRAE Interplay).**

1. PRAE never creates (partially) dead assignments.
2. PDCE may create (partially) redundant assignments.

The validity of the second part of Lemma 1 can be proven by means of the example in Fig. 9: The elimination of the partially dead occurrence of  $x := x + 1$  at node 3 creates a (total) redundancy at node 7. The validity of the first part is essentially a consequence of the following consideration. Suppose that  $s_1, s_2, \dots, s_n, s_{n+1}$  are occurrences of some assignment pattern  $\alpha$  and  $s$  an occurrence of some assignment pattern  $\beta$ . Suppose that  $s_{n+1}$  is totally redundant with respect to  $s_1, s_2, \dots, s_n$ . This implies that  $s_{n+1}$  is commonly dominated by  $s_1, s_2, \dots, s_n$  and that  $s$  becomes totally dead after eliminating  $s_{n+1}$ . This implies that the left-hand side variable of  $s$  occurs in the right-hand side term of  $s_{n+1}$  and that  $s_{n+1}$  is the last use side of this variable with respect to  $s$ . Since  $s_1, s_2, \dots, s_n$  commonly dominate  $s_{n+1}$ , there is a  $j \in \{1, \dots, n\}$  such that all paths from  $s_j$  to  $s_{n+1}$  pass statement  $s$ . This, however, contradicts the assumption that  $s_{n+1}$  is redundant.

Thus, in spite of their conceptual duality, the effects and, hence, the interplay of PDCE and PRAE are not completely dual. Though not sufficient, starting with PDCE enlarges the chance that less iterations are required in order to reach a stable state as, for example, demonstrated in Section 2, where already PDCE followed by PRAE terminates with a

locally best program.

### 5.4 Pure DAP

Pure DAP focuses on the placement of distribution assignments. Since the computational complexity of PDCE and PRAE and, hence, of DAP depends significantly on the number of iterations required for fully capturing the second-order effects of the elementary transformations, the decoupling of assignment patterns, where possible, are a major means for enhancing the algorithm's efficiency. For distribution assignments this can be achieved by a simple *preprocess* which focuses on alignments and variables occurring in distribution specifications.

**The Preprocess: Decoupling Distribution Assignment Patterns.** The starting point of the preprocess are alignments and variables occurring in distribution specifications. They can be decoupled as follows:

- *Alignments.* Alignments are replaced by explicit distribution specifications where possible. For instance, `REALIGN B(:) WITH A(:)` is replaced by `REDISTRIBUTE B(BLOCK)` if *reaching distribution analysis* yields that array `A` is distributed by `BLOCK`.
- *Variables in Distribution Specifications.* Variables used in distribution specifications, like variable `i` in `BLOCK(i)` or `CYCLIC(i)`, may prevent distribution assignment motion. *Constant propagation* helps to suspend such blockades by replacing, for example, `CYCLIC(i+j)` by `CYCLIC(3+1)` and finally `CYCLIC(4)` by expression folding.

Note that the preprocess for decoupling assignment patterns is not a prerequisite of our approach, rather a means of improving efficiency. This is important because there is, of course, a trade-off between the costs of the preprocess and the number of iterations subsequently saved. However, reaching distribution and constant propagation analysis, on which it relies, are usually exploited for other optimizations, too, and are thus in part for free.

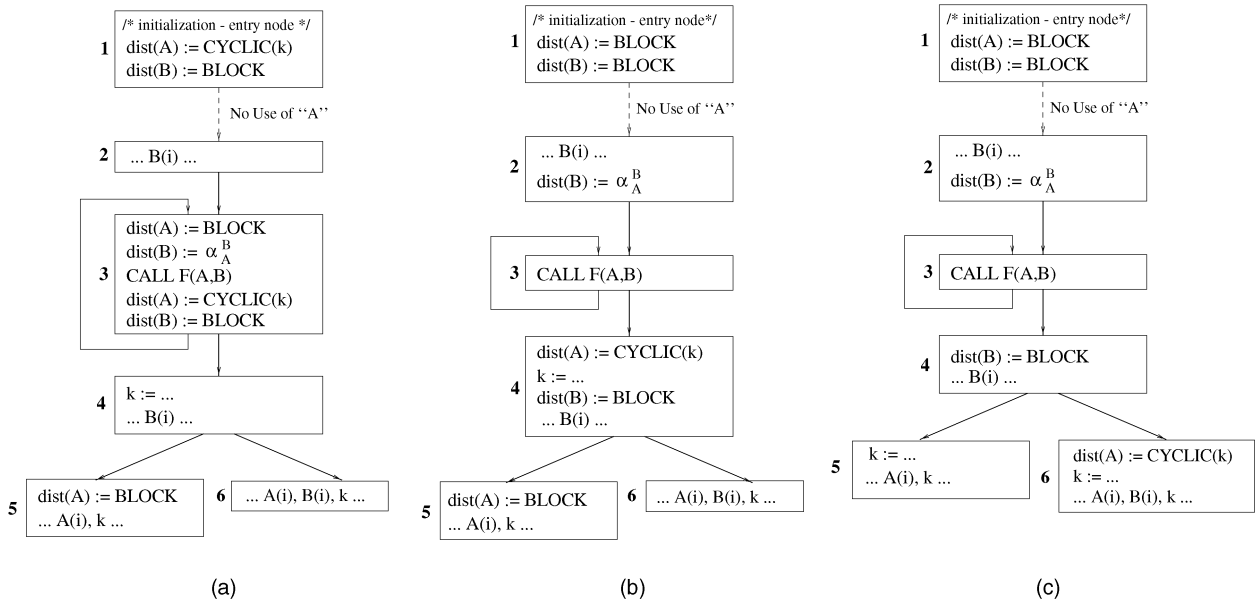


Fig. 10. Pure DAP versus Full DAP. (a) Original flow graph. (b) After Pure DAP. (c) After Full DAP.

We are now ready to present the algorithm of *Pure DAP* in detail. It is the iterated sequential composition of the algorithms for PDCE and PRAE, which have been described in detail in Sections 5.1 and 5.2, which are applied to the distribution assignment patterns of  $\mathcal{D}$  until stabilization. Following the notational convention of the previous sections and denoting the component algorithms for partially dead code elimination and partially redundant assignment elimination applied to the assignment patterns of  $\mathcal{D}$  by  $\text{PDCE}_{\mathcal{D}}$  and  $\text{PRAE}_{\mathcal{D}}$ , respectively, the following regular-expression like term defines the algorithm both completely and concisely:

$$\text{Pure DAP} : \quad \text{DAP}_{\text{pure}} \equiv (\text{PDCE}_{\mathcal{D}} \text{PRAE}_{\mathcal{D}})^+.$$

The iteration stops when both elimination steps with an interleaved assignment motion step fail to remove some code. Note, if the preprocess succeeds in decoupling all patterns of  $\mathcal{D}$ ,  $\text{PDCE}_{\mathcal{D}}$  and  $\text{PRAE}_{\mathcal{D}}$  can equivalently be replaced by the more efficient algorithms specified by  $(\text{AS}_{\mathcal{D}} \text{DCE}_{\mathcal{D}})$  and  $(\text{AH}_{\mathcal{D}} \text{RAE}_{\mathcal{D}})$ , respectively.

## 5.5 Full DAP

The *Pure DAP*-algorithm focuses on distribution assignment patterns. Second-order effects due to ordinary assignments are thus beyond its scope and not captured. As a consequence, the elimination of partially dead and partially redundant distribution assignments can get stuck by not resolving interdependences with ordinary assignments. This has been illustrated in Fig. 10b, where the distribution assignment  $\text{dist}(A) := \text{CYCLIC}(k)$  is blocked by the ordinary assignment  $k$  at node 4. In order to overcome this deadlock, the algorithm of *Full DAP* considers all assignment patterns of  $\mathcal{P}$ : It is the iterated sequential composition of the algorithms for partially dead code elimination and partially redundant assignment elimination applied to all assignment patterns of  $\mathcal{P}$ . Denoting these algorithms as in Sections 5.1 and 5.2, by  $\text{PDCE}$  and  $\text{PRAE}$ , respectively, the algorithm of *Full*

*DAP* is completely specified by the following regular-expression like term:

$$\text{Full DAP} : \quad \text{DAP}_{\text{full}} \equiv (\text{PDCE} \text{PRAE})^+.$$

Fig. 10 illustrates the different power of the algorithms of *Pure DAP* and *Full DAP* by means of the example in Fig. 5. Whereas, after *Pure DAP*, a partially dead distribution assignment remains in the program (cf. Fig. 10b), after *Full DAP* all partially dead and partially redundant distribution assignments are successfully eliminated in this example. Note, however, that also the transformationally weaker algorithm of *Pure DAP* succeeds here in removing all distribution assignments from the loop. The resulting performance improvement may already be sufficient for most practical applications.

## 5.6 Customized DAP-Variants: A Family of DAP-Algorithms

The algorithms of *Pure DAP* and *Full DAP* constitute the kernel of a family of *DAP*-algorithms of varying power and efficiency allowing a user to choose a customized *DAP*-variant, which matches his requirements or preferences. In this section, we show how to selectively enhance efficiency and power.

**Enhancing the Power.** Given a *DAP*-algorithm, its transformational power can easily be enhanced by using for ordinary assignments, instead of the partial dead-code elimination procedure, the more powerful procedure for *partial faint-code elimination* (PFCE) (cf. [16], [27]). In contrast to  $\text{PDCE}$ , PFCE eliminates assignments as “useless,” where on each program path leaving a specific program point, each right-hand side occurrence of its left-hand side variable is preceded by a modification of it or occurs in an assignment whose left-hand side variable is “faint,” too. A simple example of a faint, however, not dead statement is an assignment like  $x := x + 1$  occurring in a loop without any other occurrence of  $x$  elsewhere in the program. For

distribution patterns the notions of “dead” and “faint” coincide due to the syntactic limitations applying to their right-hand side terms. Nonetheless, in the course of exploiting second-order effects, the elimination of partially redundant and partially dead distribution assignments also profits from replacing PFCE for PDCE for ordinary assignments.

**Enhancing the Efficiency.** As pointed out, Pure DAP focuses on distribution patterns. In essence, the rationale underlying this choice is that distribution assignments are used in quite a restricted manner in practice only. Thus, it is likely to get a reasonable amount of the performance improvement of Full DAP, however, much more efficiently. In addition to limiting the statement patterns considered, this suggests to also limit the number of iterations performed. The extreme variant here is the 1-step heuristic focusing on distribution patterns expressed by the following regular-expression like term:

$$\text{DAP}_{\text{pure}}^{1\text{-step}} \equiv (\text{AS}_{\mathcal{D}} \text{DCE}_{\mathcal{D}}) (\text{AH}_{\mathcal{D}} \text{RAE}_{\mathcal{D}}).$$

$\text{DAP}_{\text{pure}}^{1\text{-step}}$  applies the elementary transformations of assignment sinking, dead code elimination, assignment hoisting, and redundant assignment elimination only once. It thus does not capture any second-order effects, preventing to prove any meaningful formal optimality criterion for it. However, it is extremely efficient and often very effective in practice; e.g., for the ADI example in Fig. 6 and the example in Fig. 4, the above 1-step heuristic yields the optimal results and, for the example in Fig. 5, it succeeds in removing all distribution assignments from the loop.

**Masking Distribution Assignments.** Each DAP-algorithm of our family can be combined with *distribution assignment masking*. This means to prefix distribution assignments by masks ensuring that they are only performed if the actual distribution differs from the distribution which would be assigned to by the statement under consideration. Technically, a distribution assignment of the form  $\text{dist}(\mathbf{A}) := \delta$  is replaced by a conditionally executed one of the form:  $\text{if } (\text{dist}(\mathbf{A}) \neq \delta) \text{ dist}(\mathbf{A}) := \delta$ .

Evaluating a mask consists of determining  $\delta$  and comparing it with the actual distribution of array  $\mathbf{A}$ . The effort required depends on the concrete implementation of the runtime system and may differ significantly for regular distributions like BLOCK and irregular distributions with mapping arrays, but usually it is much cheaper than the distribution statement it guards. Thus, masking can be used to reduce the overhead caused by partially redundant distribution assignments remaining in a program after DAP. Note, however, that masking introduces overhead for program executions for which the masked occurrence is not redundant. Moreover, masking does not apply to remaining partially dead assignments: they have to be executed in any case.

**Uniform Integration of Output Statements and Conditions.** In practice, also second-order effects due to output statements and conditions must be taken into account. Enhancing our approach accordingly is actually straightforward. It suffices to consider *output statements* as assignments with no left-hand sides. This allows us to handle them uniformly in our approach. Similarly, this can

be achieved for *conditions*. Replace each condition by an assignment to a fresh variable followed by the branch statement with the original condition replaced by the fresh variable. Obviously, the fresh variable does not block any assignment, just as it did not occur in the original program. On the other hand, the statement added can be subjected to PDCE and PRAE like any other statement. Pragmatically, it is advantageous to associate lexically identical conditions with the same fresh variable in order to open opportunities for redundancy eliminations.

## 5.7 Main Results: Optimality

Let  $G_{\text{pure}}$  and  $G_{\text{full}}$  denote the programs, which result from our algorithms  $\text{DAP}_{\text{pure}}$  and  $\text{DAP}_{\text{full}}$  applied to a program  $G$ , respectively. Additionally, let

$$\mathcal{G}_{\text{mix}_{\mathcal{D}}} =_{df} \{G' \mid G \vdash_{(\text{AS}_{\mathcal{D}}+\text{DCE}_{\mathcal{D}}+\text{AH}_{\mathcal{D}}+\text{RAE}_{\mathcal{D}})}^* G'\}$$

and

$$\mathcal{G}_{\text{mix}} =_{df} \{G' \mid G \vdash_{(\text{AS}+\text{DCE}+\text{AH}+\text{RAE})}^* G'\}$$

denote the sets of programs derivable from  $G$  by repeated applications of the four elementary transformations of Pure DAP and Full DAP in any order. Then we have:

**Theorem 1 (1st Optimality Theorem).**  $G_{\text{pure}}$  and  $G_{\text{full}}$  are locally best in  $\mathcal{G}_{\text{mix}_{\mathcal{D}}}$  and  $\mathcal{G}_{\text{mix}}$ , respectively, i.e., they cannot be improved any further by means of admissible assignment sinkings (hoistings) or dead (redundant) assignment eliminations.

Actually, this is the best we can expect for the result of Pure and Full DAP with respect to the universes of programs  $\mathcal{G}_{\text{mix}}$  ( $\mathcal{G}_{\text{mix}_{\mathcal{D}}}$ ) in general. As illustrated in Section 5.3,  $\mathcal{G}_{\text{mix}}$  ( $\mathcal{G}_{\text{mix}_{\mathcal{D}}}$ ) lacks in general the existence of a program being “globally best,” but provides a number of programs being “locally best,” i.e., which cannot be improved any further by means of the component transformations of PDCE and PRAE. In particular, any remaining partially dead or partially redundant assignment in  $G_{\text{full}}$  ( $G_{\text{pure}}$ ) cannot be removed by the respective class of assignment sinkings/hoistings and dead/redundant assignment eliminations under consideration without modifying the branching structure of the program or impairing some program executions.

In practice, even the simple sequential composition of PDCE and PRAE without iterating is often sufficient to remove all partially dead and partially redundant assignments. Though this does not hold in general (in particular, the programs  $G_{\text{pure}}^{\text{simpl}}$  and  $G_{\text{full}}^{\text{simpl}}$  resulting from the composition of PDCE and PRAE applied to  $G$  need not to be locally best in  $\mathcal{G}_{\text{mix}_{\mathcal{D}}}$  and  $\mathcal{G}_{\text{mix}}$ , respectively), the following optimality result applies to them. Let

$$\mathcal{G}_{\text{pure}} =_{df} \{G' \mid G \vdash_{(\text{AS}_{\mathcal{D}}+\text{DCE}_{\mathcal{D}})}^* \bar{G} \vdash_{(\text{AH}_{\mathcal{D}}+\text{RAE}_{\mathcal{D}})}^* G'\}$$

and

$$\mathcal{G}_{\text{full}} =_{df} \{G' \mid G \vdash_{(\text{AS}+\text{DCE})}^* \bar{G} \vdash_{(\text{AH}+\text{RAE})}^* G'\},$$

where  $\bar{G}$  is assumed to be invariant under further assignment sinkings (up to local reorderings in basic blocks) and dead code eliminations. Then, we have (cf. [27], [28]):

<pre> ! main program program P real A(N,N) ... ! assumed: A ≡ (block,*) ! start measurement do iter=1,maxiter   dist(A) := (*,block)   call F1(A,N)   dist(A) := (block*) end do ! stop measurement ... end </pre>	<pre> ! internal subprogram subroutine F1(X,N) real X(N,N) !HPF\$ distribute X(*,block) do j=1,N   do i=2,N-1     X(i,j)=X(i,j)/2.0 + (X(i-1,j)+ +      X(i+1,j))/4.0   end do end do end </pre>
(a)	
<pre> ! main program program P real A(N,N) ... ! assumed: A ≡ (block,*) ! start measurement dist(A) := (*,block) do iter=1,maxiter   call F1(A,N) end do dist(A) := (block,*) ! stop measurement ... end </pre>	<pre> ! internal subprogram subroutine F1(X,N) real X(N,N) !HPF\$ distribute X(*,block) do j=1,N   do i=2,N-1     X(i,j)=X(i,j)/2.0 + (X(i-1,j)+ +      X(i+1,j))/4.0   end do end do end </pre>
(b)	

Fig. 11. Basic and optimized program versions of an HPF code fragment. (a) Basic distribution assignment generation. (b) Result of DAP.

**Theorem 2 (2nd Optimality Theorem).**  $G_{full}^{smp}$  and  $G_{pure}^{smp}$  are optimal in  $\mathcal{G}_{full}$  and  $\mathcal{G}_{pure}$ , respectively.

## 6 PERFORMANCE MEASUREMENTS

In this section, we report on practical experiences showing the effectivity of DAP by measuring the runtimes of programs before and after our optimization. As hardware platforms, a Quadrics CS-2 (formerly Meiko CS-2), a NEC Cenju-4, and a Beowulf cluster are used. Whereas the first two platforms are distributed memory, massively parallel supercomputers with a dedicated communication network, and a good computation/communication ratio, the Beowulf cluster is connected via Fast Ethernet requiring high data locality to achieve good performance.

The programs have been compiled with our vfc system and with vfc [40], the successor system of vfc. Both compilation systems perform a source-to-source translation from HPF programs to explicitly parallel Fortran message passing codes. vfc provides efficient compilation strategies and powerful optimizations for regular codes, but does not support irregular codes appropriately. To overcome these deficiencies, we started the development of vfc, putting the main emphasis on irregular codes. The parallelization strategy of both compilation systems is based upon the

Single-Program-Multiple-Data (SPMD) paradigm. In that model, each processing node executes the same (parameterized) code in a loosely synchronous fashion. The generated SPMD programs call MPI routines [13] to realize the data transfer between the nodes of a parallel machine. In addition, we use in our numerical experiment ScaLAPACK [3], a wide-spread high-performance distributed memory linear algebra package which stands for Scalable LAPACK [2].

In our first experiment, we investigate the potential of our optimization. We measure how much it costs to remap an array compared with the costs to iterate over an array and to perform a stencil operation. If the computation time is very high compared with the communication time, the relative performance improvement will be limited for this kind of optimization. On the other hand, if the communication time is very high, considerable performance improvements can be achieved. The measurements give an impression on the fraction of time required to perform calculations on whole arrays and the time required to redistribute whole arrays. Fig. 11 shows the original and optimized version of an HPF code consisting of a stencil operation which accesses a prescriptively mapped array. Subroutine F1 sweeps over the columns calculating new

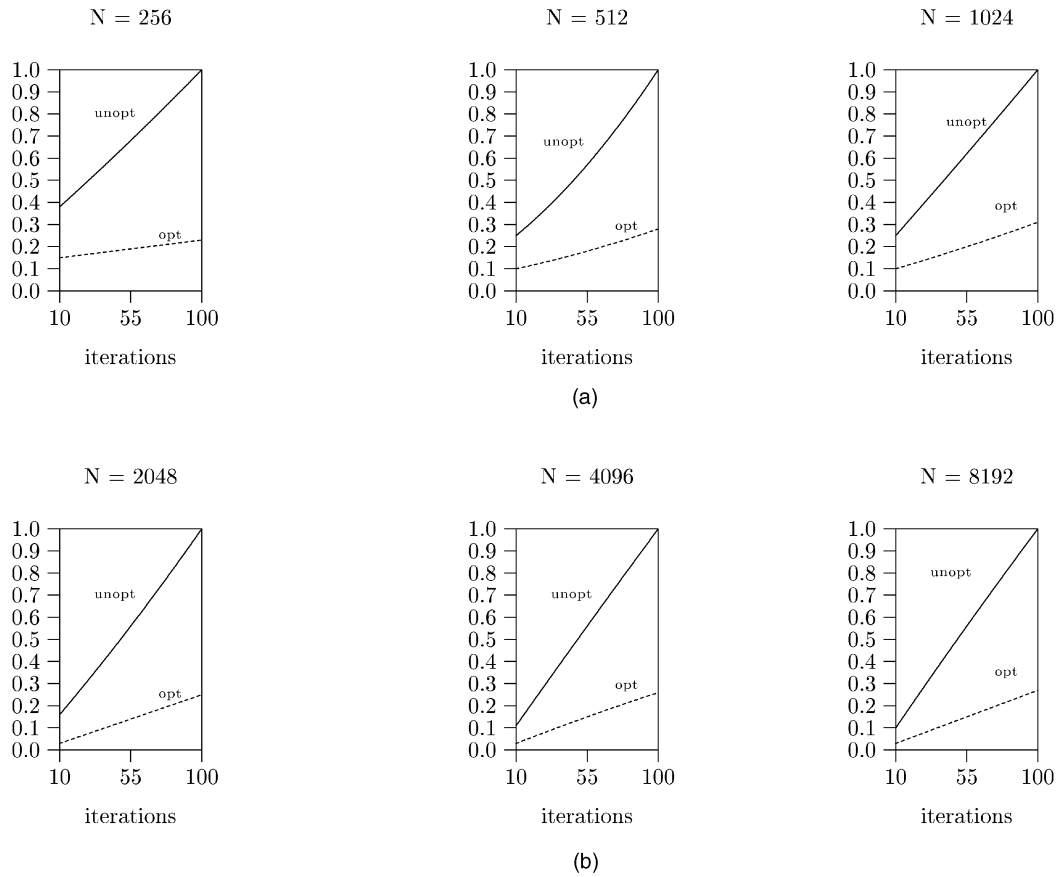


Fig. 12. Normalized runtimes for program shown in Fig. 11 on a Quadrics CS-2 and a NEC Cenju-4. (a) Quadrics CS-2 with 16 processors. (b) NEC Cenju-4 with 32 processors.

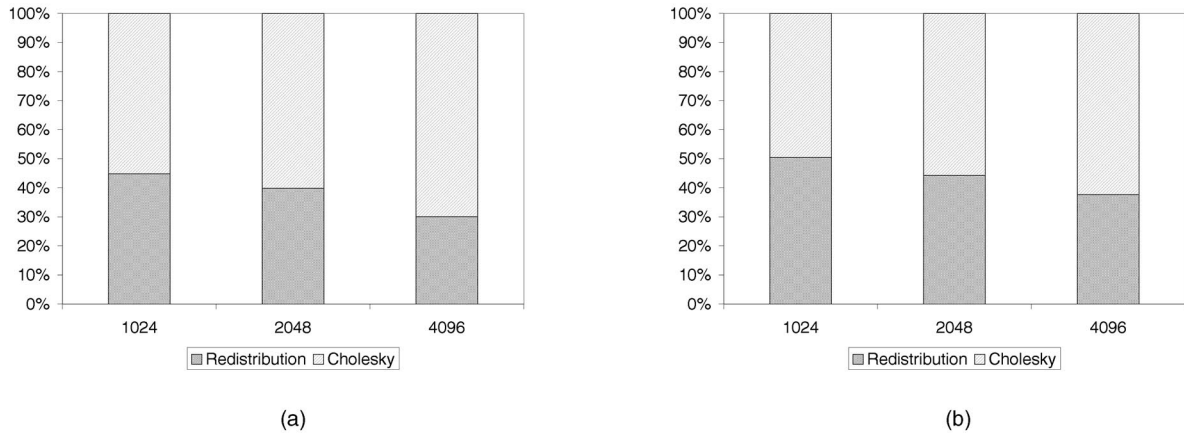


Fig. 13. Effort for Cholesky factorization and redistributions. (a) Cluster with eight nodes. (b) Cluster with 16 nodes.

values for  $X(i, j)$  by looking at the neighboring elements  $X(i-1, j)$  and  $X(i+1, j)$ . Choosing a column-wise mapping, no communication and synchronization is required within the sweeps. In our example, it is assumed that array  $A$  is distributed in the main program by (block,\*). The prescriptively mapped array results in the insertion of distribution assignments at the call site, as shown in Fig. 11a. DAP succeeds in moving both mapping assignments out of the loop as shown in Fig. 11b. The execution times of the original and optimized program versions are shown in Fig. 12. We ran both programs in single-precision

mode with problem sizes 256, 512, and 1,024 on the CS-2 with 16 processors and with problem sizes 2,048, 4,096, and 8,192 on the Cenju-4 with 32 processors. The x-axis of the chart specifies the number of iterations; the y-axis is normalized such that the longest execution time is denoted by time unit 1.0 facilitating comparisons between the charts.

On the CS-2, a performance improvement could be achieved from 153 percent to 335 percent for  $maxiter = 10$  to  $maxiter = 100$ , for problem size 256; for problem sizes 512 and 1,024, the performance improvement was between 150 percent and 257 percent for  $maxiter = 10$  to

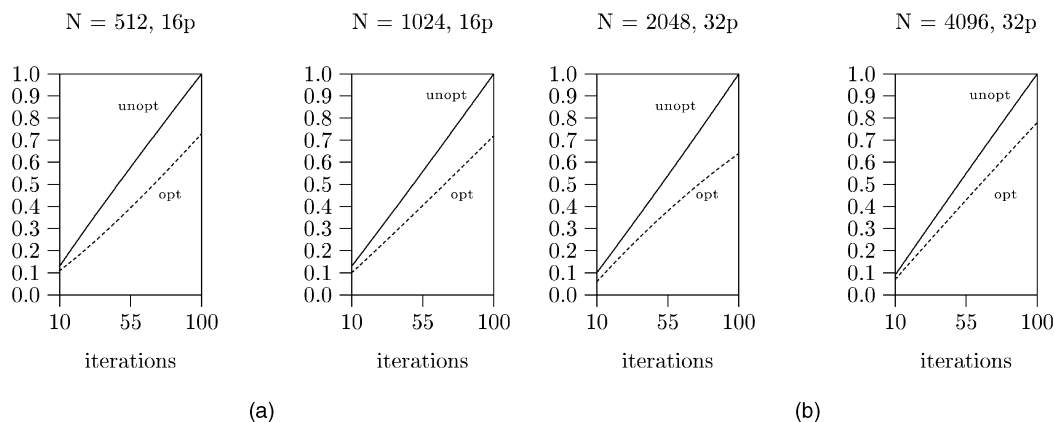


Fig. 14. Normalized runtimes for ADI integration kernel with various problem sizes and number of processors for different hardware platforms. (a) Quadrics CS-2. (b) NEC Cenju-4.

$maxiter = 100$ . On the Cenju-4, the performance improvement reaches for  $N = 2,048$  about 380 percent and, for problem sizes 4,096 and 8,192, about 280 percent. Although we are using different hardware platforms, different compilers with different runtime systems, and different problem sizes and numbers of processing nodes, the results are similar and show the potential for this kind of optimization. The dramatic improvement observed, however, is not surprising in the light of average remapping times (cf. [39], [20], [35], [6]). More expensive stencil operations and remappings remaining in the loop, however, will shrink the performance improvement (cf. Fig. 14).

Note that kernels like the program fragment of Fig. 11 are likely to occur in numerical applications when numerical library routines are called from HPF. In [10], it has been shown that existing numerical libraries can be integrated into HPF code in a portable and efficient way, yielding significant performance improvements compared to reimplementations based on HPF/Fortran which do not take advantage of highly-optimized packages. As a consequence, interfaces between HPF and numerical libraries have been developed. SLHPF [4] provides an interface from HPF to ScaLAPACK [3]. Since ScaLAPACK compatible distributions are required, remappings may occur at subroutine boundaries (see examples in [10]) which may worsen performance significantly [9]. WIEN97 [5], a sophisticated software package used for quantum mechanical calculations of solids and developed at the Institute of Physical and Theoretical Chemistry at the Vienna University of Technology, spends about 40 to 95 percent in numerical library routines which are called throughout the program. One typical representative of those library routines with cubic complexity is the Cholesky factorization which plays in the WIEN97 code an important role as well. Fig. 13 shows the fraction of time it takes to establish a new mapping at entry of the call and to restore the original one at exit compared with the time required to perform the factorization. The measurement is done on a Beowulf cluster with eight and 16 processors for problem sizes 1,024, 2,048, and 4,096 using double-precision. Although the fraction of time required for remappings is shrinking for larger problem sizes, the elimination results in a considerable

improvement and is important for integrating numerical libraries in HPF.

In our final experiment, we measure the runtimes of the unoptimized and optimized ADI integration kernel described in Section 2 and shown in Fig. 6. Remember that we succeeded in reducing the number of remappings required in one loop iteration from 12 to 6. We ran our tests on the CS-2 and Cenju-4 with 16 and 32 processing nodes with single-precision arrays for problem sizes 512, 1,024, 2,048, and 4,096 with the number of iterations ranging from  $maxiter = 10$  to  $maxiter = 100$ . The normalized runtimes on the CS-2 are shown in Fig. 14a for 16 processors and problem sizes 512 and 1,024. The results on the Cenju-4 are reported in Fig. 14b for problem sizes 2,048 and 4,096 on 32 processors.

On the CS-2, we achieved a performance improvement on 16 processing nodes for both problem sizes from about 24 percent for 10 iterations to 38 percent for 100 iterations. On the Cenju-4 the performance improvement is about 55 percent for problem size 2,048 and 30 percent for problem size 4,096. Although the benefit is shrinking with increasing problem size, the performance improvement is nevertheless considerable.

## 7 CONCLUSIONS

Inefficient placement of redistributions has been identified by several researchers as a major source of unsatisfying performance. Previous approaches to improve on this situation, however, were only aiming at eliminating totally dead or (partially) redundant redistributions. As demonstrated in this paper, this is in general insufficient. In fact, generally, the combined effect of eliminating both *partially dead* and *partially redundant* redistributions is required in order to get the full benefit.

Based on the algorithms for PDCE and PRAE of [27] and [28], which have been designed for standard sequential programs, we showed how to adapt and combine them to optimize data remappings in data-parallel languages.<sup>5</sup> Second-order effects between PDCE and PRAE showing up by combining them introduced the need of a refined

5. Note that performing special loop optimizations like loop invariant code motion in addition to elimination of dead and redundant distribution changes is less powerful than the combined effect of PDCE and PRAE without saving any implementational effort.

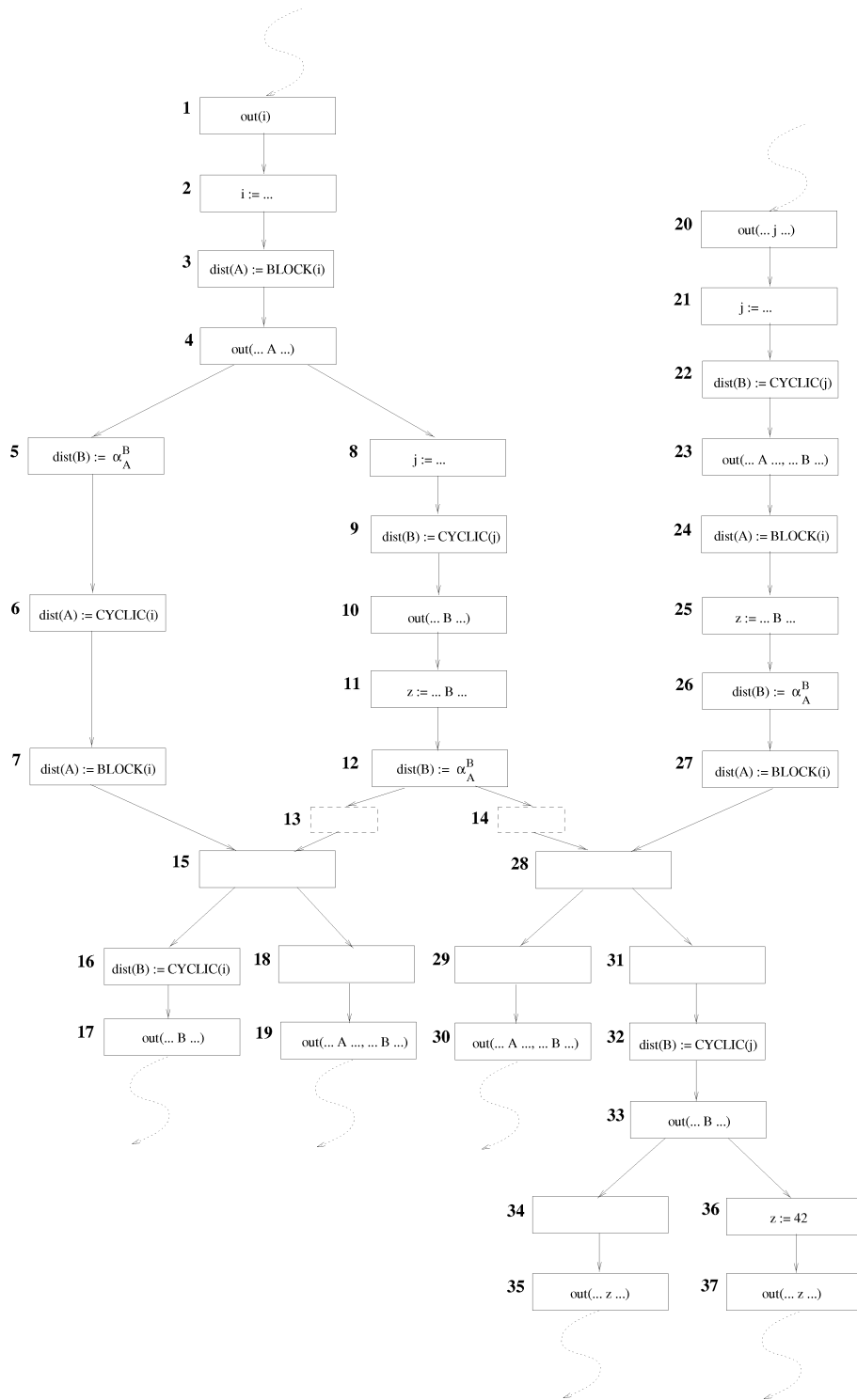


Fig. 15. The original flow graph.

optimality investigation, a result which is of common interest in compiler optimizations since partial dead-code elimination and partial redundancy elimination are usually applied to ordinary assignments in sequential compilers as well. The refined optimality investigation led to a family of algorithms for distribution assignment placement of varying power and efficiency, offering customized solutions according to a user’s requirements or preferences. This

ranges from extremely efficient one-step heuristics to extremely powerful procedures like the enhanced Full DAP-algorithm, uniformly resolving all second-order effects between all assignment patterns (ordinary and distribution assignments). We hope that this encourages programmers to make use of prescriptively mapped arrays or REDISTRIBUTE directives, where it is the natural means for expressing their needs without being

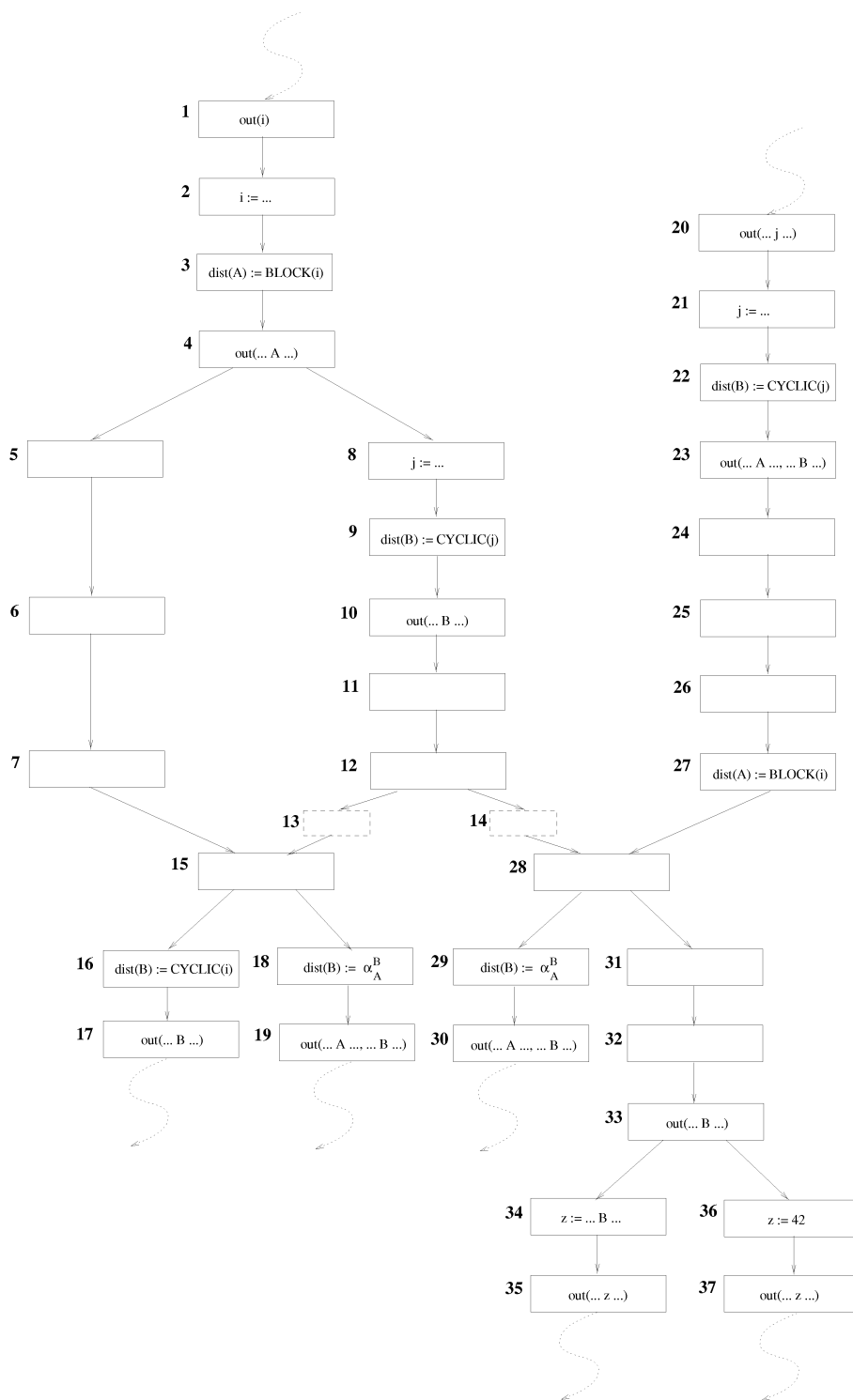


Fig. 16. Stable after PDCE-PRAE-PDCE-PRAE-PDCE.

worried of having to avoid introducing superfluous redistributions, as they can be assured that unnecessary redistributions will be eliminated automatically, thus supporting a problem-centered programming style. Actually, a programmer must rely on the fact that compilers support the language standard properly.

Finally, it is worth noting that our approach is not limited to distributed-memory systems. In fact, data locality

is also an important issue for nonuniform memory access (NUMA) architectures like virtual shared memory multiprocessors. Moreover, we are currently investigating the practical impact of an interprocedural extension of our approach (cf. [23]), based on [29], and how it compares to both its intraprocedural counterpart and other interprocedural algorithms.

## APPENDIX

### ITERATED SECOND-ORDER EFFECTS OF PDCE AND PRAE

In this section, we illustrate the impact of second-order effects of PDCE and PRAE. To this end, we consider the example in Fig. 15. Starting with PDCE, the repeated interleaved application of PRAE and PDCE gets stable only after four additional applications of PRAE and PDCE, respectively. The program finally resulting from this process is shown in Fig. 16. The intermediate transformation results can be found in [22], where this example is discussed in full detail.

### ACKNOWLEDGMENTS

The authors would like to thank GMD TechnoPark and National Electronic Code (NEC Europe Ltd.) C&C Research Laboratories, both located in Sankt Augustin, Germany, for giving them the opportunity to run their experiments on a NEC Cenju-4 and for their assistance. Further, they would like to thank Dieter Kvasnicka for his assistance in the numerical example. Finally, the comments of the anonymous referees were very helpful in improving the article. This research was partially supported by the Austrian Science Fund as part of Aurora Project "Languages and Compilers for Scientific Computation" under Contract SFB-011.

### REFERENCES

- [1] G. Agrawal, J. Saltz, and R. Das, "Interprocedural Partial Redundancy Elimination and Its Application to Distributed Memory Compilation," *Proc. ACM SIGPLAN '95 Conf. Programming Language Design and Implementation (PLDI '95)*, vol. 30, no. 6, pp. 258-269, 1995.
- [2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide, Release 1.0*. Philadelphia: SIAM, 1992.
- [3] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, *Scalpack Users' Guide*. 1997.
- [4] L.S. Blackford, J.J. Dongarra, C.A. Papadopoulos, and R.C. Whaley, "Installation Guide and Design of the HPF 1.1 Interface to ScaLAPACK, SLHPF," Technical Report CS-98-396, Univ. of Tennessee, Aug. 1998.
- [5] P. Blaha, K. Schwraz, G. Madsen, D. Kvasnicka, and J. Luitz, "WIEN97/WIEN2k," Institute of Materials Chemistry, TU Vienna. <http://www.wien2k.at/index.html>, 2002.
- [6] Y.-C. Chung, C.-H. Hsu, and S.-W. Bai, "A Basic-Cycle Calculation Technique for Efficient Dynamic Data Redistribution," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 4, Apr. 1998.
- [7] F. Coelho and C. Ancourt, "Optimal Compilation of HPF Remappings," *J. Parallel and Distributed Computing*, vol. 38, no. 2, pp. 229-236, Nov. 1996.
- [8] D.M. Dhamdhere and H. Patil, "An Elimination Algorithm for Bidirectional Data Flow Problems Using Edge Placement," *ACM Trans. Programming Languages and Systems*, vol. 15, no. 2, pp. 312-336, Apr. 1993.
- [9] H.J. Ehold, W.N. Gansterer, D.F. Kvasnicka, and C.W. Ueberhuber, "High Local Performance in HPF Codes," Technical Report Aurora TR2000-06, Inst. for Software Science, Univ. of Vienna, Austria, 2000. Electronically available at <http://www.vpcp.univie.ac.at/aurora/publications/>.
- [10] H.J. Ehold, W. Gansterer, D.F. Kvasnicka, and C.W. Ueberhuber, "HPF and Numerical Libraries," *Proc. Fourth Int'l ACPC Conf.*, Feb. 1999.
- [11] T. Fahringer and E. Mehofer, "Buffer-Safe and Cost-Driven Communication Optimization," *J. Parallel and Distributed Computing*, vol. 57, pp. 33-63, 1999.
- [12] High Performance Fortran Forum "High Performance Fortran Language Specification Version 2.0," technical report, Rice Univ., Houston, TX, Jan. 1997. Available via HPFF home page: <http://www.crpc.rice.edu/HPFF>.
- [13] Message Passing Interface Forum *MPI-2: Extensions to the Message-Passing Interface*. July 1997.
- [14] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, "FORTRAN D Language Specification," technical report, Rice Univ., Houston, TX, Jan. 1992.
- [15] A. Geser, J. Knoop, G. Lüttgen, O. Rüthing, and B. Steffen, "Non-Monotone Fixpoint Iterations to Resolve Second Order Effects," *Proc. Sixth Int'l Conf. Compiler Construction (CC '96)*, pp. 106-120, 1996.
- [16] R. Giegerich, U. Möncke, and R. Wilhelm, "Invariance of Approximative Semantics with Respect to Program Transformations," *Proc. Third Conf. European Co-Operation in Informatics, Informatik-Fachberichte*, vol. 50, pp. 1-10, 1981.
- [17] M. Gupta, E. Schonberg, and H. Srinivasan, "A Unified Data-Flow Framework for Optimizing Communication," *Proc. Seventh Workshop Languages and Compilers for Parallel Computing*, Aug. 1994.
- [18] M.W. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines," *Proc. Supercomputing '92*, pp. 522-534, Nov. 1992.
- [19] M.S. Hecht, *Flow Analysis of Computer Programs*. North-Holland: Elsevier, 1977.
- [20] S.D. Kaushik, C.H. Huang, J. Ramanujam, and P. Sadayappan, "Multi-Phase Array Redistribution: Modeling and Evaluation," *Proc. Int'l Parallel Processing Symp.*, 1995.
- [21] K. Kennedy and A. Sethi, "A Constraint Based Communication Placement Framework," Technical Report CRPC-TR95515-S, Dept. of Computer Science, Rice Univ., Houston, TX, Feb. 1995.
- [22] J. Knoop and E. Mehofer, "Distribution Assignment Placement: A New Aggressive Approach for Optimizing Redistribution Costs," Technical Report TR 97-6, Inst. for Software Science, Univ. of Vienna, Austria, 1997.
- [23] J. Knoop and E. Mehofer, "Interprocedural Distribution Assignment Placement: More than just Enhancing Intraprocedural Placing Techniques," *Proc. Fifth IEEE Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '97)*, pp. 26-37, 1997.
- [24] J. Knoop and E. Mehofer, "Optimal Distribution Assignment Placement" *Proc. Third European Conf. Parallel Processing (Euro-Par '97)*, vol. 1300, pp. 364-373, 1997.
- [25] J. Knoop, O. Rüthing, and B. Steffen, "Lazy Code Motion," *Proc. ACM SIGPLAN '92 Conf. Programming Language Design and Implementation (PLDI '92)*, vol. 27, no. 7, pp. 224-234, 1992.
- [26] J. Knoop, O. Rüthing, and B. Steffen, "Optimal Code Motion: Theory and Practice," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 4, pp. 1117-1155, 1994.
- [27] J. Knoop, O. Rüthing, and B. Steffen, "Partial Dead Code Elimination," *Proc. ACM SIGPLAN '94 Conf. Programming Language Design and Implementation (PLDI '94)*, vol. 29, no. 6, pp. 147-158, 1994.
- [28] J. Knoop, O. Rüthing, and B. Steffen, "The Power of Assignment Motion," *Proc. ACM SIGPLAN '95 Conf. Programming Language Design and Implementation (PLDI '95)*, vol. 30, no. 6, pp. 233-245, 1995.
- [29] J. Knoop, O. Rüthing, and B. Steffen, "Towards a Tool Kit for the Automatic Generation of Interprocedural Data Flow Analyses," *J. Programming Languages*, vol. 4, no. 4, pp. 211-246, 1996.
- [30] U. Kremer, "Automatic Data Layout for Distributed Memory Machines," PhD thesis, Rice Univ., Houston, TX, Oct. 1995.
- [31] E. Mehofer and H. Zima, "Distribution Assignment Placement," Technical Report TR-96-5, Inst. for Software Science, Univ. of Vienna, Austria, 1996.
- [32] S.S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco: Morgan Kaufmann, 1997.
- [33] NPAC "HPF Applications Kernels," Northeast Parallel Architectures Center, Syracuse Univ., NY, <http://www.npac.syr.edu/hpfa/>, Nov. 1996.
- [34] D.J. Palermo, E.W. Hodges, and P. Banerjee, "Interprocedural Array Redistribution Data-Flow Analysis," *Proc. Ninth Workshop Languages and Compilers for Parallel Computing*, pp. 435-449, Aug. 1996.

- [35] S. Ramaswamy, B. Simons, and P. Banerjee, "Optimizations for Efficient Array Redistribution on Distributed Memory Multi-computers," *J. Parallel and Distributed Computing*, vol. 38, no. 2, pp. 217-228, Nov. 1996.
- [36] S. Ranka, H.W. Yau, K.A. Hawick, and G.C. Fox, "High-Performance Fortran for SPMD Programming: An Applications Overview," Technical Report SCCS-805, Syracuse Univ., NPAC, Syracuse, NY, May 1997.
- [37] B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "Global Value Numbers and Redundant Computations," *Conf. Record 15th ACM Symp. Principles of Programming Languages (POPL '88)*, pp. 2-27, 1988.
- [38] B. Steffen, "Property-Oriented Expansion," *Proc. Third Static Analysis Symp. (SAS '96)*, pp. 22-41, 1996.
- [39] R. Thakur, A. Choudhary, and J. Ramanujam, "Efficient Algorithms for Array Redistribution" *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 6 pp. 587-594, June 1996.
- [40] VFCS/VFC Webpage, Inst. for Software Science, Univ. of Vienna, <http://www.par.univie.ac.at/research/research-projects.html>, 2002.
- [41] H.W. Yau, G.C. Fox, and K.A. Hawick, "Evaluation of High Performance Fortran through Application Kernels," *Proc. High-Performance Computing and Networking (HPCN '97)*, Apr. 1997.
- [42] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, "Vienna Fortran—A Language Specification Version 1.1," *Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation*, Mar. 1992.



**Jens Knoop** received the Diploma and PhD degrees in computer science from the University of Kiel, Germany, in 1987 and 1993, respectively. He is currently a member of the programming systems and compiler construction group at the University of Dortmund, Germany. His primary research interests are in the areas of analysis and verification of programs and systems including optimizing and parallelizing compilers, abstract interpretation, and model checking. He is the general chair of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI) and a member of the IEEE Computer Society and the ACM.



**Eduard Mehofer** received the Diploma degree in computer science from the Vienna University of Technology in 1989. In 1998, he received the PhD degree in computer science from the Vienna University of Technology. From 1986 to 1993, he was employed at the Alcatel Research Center, Vienna, and worked in the field of expert system languages. In 1993, he joined the Institute of Software Science, headed by Professor Zima, at the University of Vienna.

Currently, he is an assistant professor at the Institute of Software Science, University of Vienna. His research focuses on parallel and distributed computing and on compilation techniques, in particular, communication optimizations, feedback-directed compilation, and data flow frequency analysis. He is a member of the IEEE Computer Society.

► **For more information on this or any computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.**