A Powerful and Flexible Approach for Distribution Assignment Placement

Jens Knoop * Eduard Mehofer †

Abstract

Dynamic data redistributions in data-parallel languages like HPF address the demands posed by advanced applications with different computational kernels or dynamically varying processor workloads and are a major means for improving the performance. On the other hand, redistributions can be very expensive and significantly degrade a program's performance. In this article, we present a powerful and flexible approach for avoiding unnecessary remappings by eliminating *partially dead* and *partially redundant* distribution changes. Basically, this approach evolves from extending and combining two algorithms for these optimizations for sequential programs. In contrast to the sequential setting, the data-parallel setting leads to a hierarchy of algorithms of varying power and efficiency. The flexibility resulting from this makes it easy to trade efficiency against power according to a user's requirements. The approach is demonstrated by several illustrating examples.

1 Motivation

The user-controlled distribution of data across the local memories of the processing nodes is a central feature of data-parallel languages like *High Performance Fortran* (HPF) [3], *Fortran D* [4], or *Vienna Fortran* [13]. A program's performance can critically depend on the distribution chosen. Dynamic data redistributions, e.g., in case of varying computational kernels or dynamically varying processor workloads, are thus a major means for improving the performance. On the other hand, remappings can be quite expensive as communication is required to migrate the array elements to their new owning processors. Unnecessary distribution changes can therefore cause a significant loss of performance. Avoiding them is of key importance to gain efficiency.

In this article we present a powerful and flexible approach for *distribution assignment placement* (DAP), which allows customized solutions according to a user's requirements and preferences on efficiency and power. In essence, the new approach works by eliminating *partially dead* and *partially redundant* distribution changes. Basically, it evolves from extending and combining two algorithms for these optimizations for sequential programs (cf. [7, 8]). Intuitively, it computes beneficial insertion points for distribution assignments by means of code *hoisting* and *sinking* interleaved by eliminating *redundant* and *dead* code, which uniformly captures removal of unnecessary remappings in straight-line code as well as in loops. This is illustrated in Figure 1. Important are the distribution assignments occurring inside the loop. Distribution assignments are inserted by the compiler whenever an array is associated with a distribution.[1] In the example of Figure 1 the arrays A and B are assumed to be statically distributed by BLOCK. The distribution assignments dist(A) := BLOCK (CYCLIC) indicate that the distribution of A

*Universität Passau, Fakultät für Mathematik und Informatik, Innstraße 33, D-94032 Passau, Germany (e-mail: knoop@fmi.uni-passau.de).

†Universität Wien, Institut für Softwaretechnik und Parallele Systeme, Liechtensteinstraße 22, A-1090 Vienna, Austria (e-mail: mehofer@par.univie.ac.at).

[1] HPF allows to change the distribution of arrays explicitly as well as implicitly with *redistribute/realign directives* and at subprogram boundaries, respectively. Distribution assignments, introduced in [10] as part of the intermediate program code, allow a uniform treatment of all remappings.

is changed to BLOCK and CYCLIC, respectively (analogously for B), which are inserted in order to establish the requested distributions on entry and to restore the original ones on exit of the subroutine call according to the HPF semantics [10]. Note that none of the distribution assignments is (totally) redundant or dead. This prevents to simply eliminate them. However, the distribution assignments after leaving subroutine F are *partially dead* because they are dead within the loop (and alive with respect to node **3**). Thus, they can be removed from the loop as depicted in Figure 1(b). As a side-effect, this turns the totally live distribution assignments before entering subroutine F, which before were required on every loop iteration, to be *partially redundant*. They are redundant within the loop (but not wrt the static distribution), and, hence, can be hoisted out of the loop as shown in Figure 1(b). Such effects are usually called *second-order effects*. Whereas second-order effects due to interdependences of different assignment patterns are well-known, and can as usual be overcome by repeated applications of the elementary transformations, the interleaving of all four elementary transformations of PDCE and PRAE reveals additional interdependences of the transformations themselves. As a consequence, the program finally resulting depends on the specific sequence of the elementary transformations, which is in contrast to PDCE and PRAE. On the other hand, and unlike the sequential setting, the data-parallel setting leads to a hierarchy of algorithms of varying power and efficiency fitting a user's individual needs. This is demonstrated by means of the algorithms of *Pure* DAP and *Full* DAP, and several refinements based thereof. Whereas Pure DAP focusses on distribution assignments and resolves second-order effects among them, Full DAP resolves even all second-order effects between ordinary and distribution assignments. Partially dead and partially redundant (distribution) assignments remaining in the program after Full DAP cannot be eliminated further without changing its branching structure or impairing some of its executions.

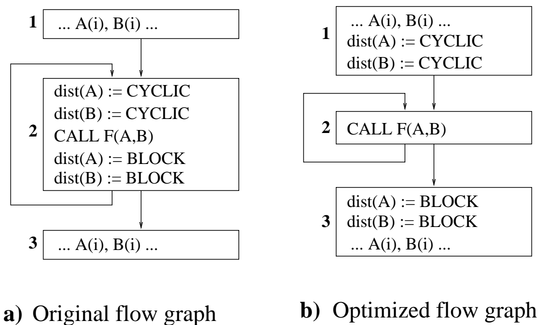


Figure 1: Small example

Related work: Redistribution analysis has been addressed by several researchers. Hall et al. [5] presented techniques for hoisting remappings out of loops and eliminating dead remappings. Their approach is incomparable to ours as it takes interprocedural information into account, but does not consider the more general problem of eliminating partially dead and partially redundant remappings. Coelho et al. [2] describe an optimization which reduces the communication amount by removing useless remappings and taking advantage of replications to shorten individual remappings. Reducing the overall number of remappings by employing code motion is not addressed. Similarly, this holds for Ramaswamy et al. [12], whose focus lies on the automatic generation of efficient routines for migrating the array elements to their new owning processors. Palermo et al. [11] present several analyses related to dynamic redistributions: computation of reaching distributions, making all remappings (including the implicit ones at subprogram boundaries) explicit and removing the redundant ones, and converting programs with dynamically distributed arrays into subset HPF. Motion of data remappings is not considered.

2 Preliminaries

As usual we represent a program by a *directed flow graph* $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N and edge set E. Nodes $n \in N$ represent *basic blocks* of instructions, edges $(m, n) \in E$ the nondeterministic branching structure of G, and $\mathbf{s}$ and $\mathbf{e}$ the unique *start node* and *end node* of G, which are assumed to have no incoming and outgoing edges, respectively. All statements of a program

are classified as follows: (ordinary) *assignment statements* including both scalar and indexed variables; the *empty statement* skip; *distribution assignments* of the form $dist(A) := \delta$, which are generated by the compiler and uniformly express distribution changes occurring throughout the program at subprogram boundaries and for redistribute/realign directives; subprogram calls, and output operations of the form $out(t)$ forcing all operands of term t to be alive. Finally, all edges leading from a node with several successors to a node with several predecessors are assumed to be split by a synthetic node. This is typical for code motion transformations in order to avoid that the motion process gets stuck (cf. [7, 8]).

3 Distribution Assignment Placement

In this section we stepwise develop our hierarchy of DAP-algorithms for eliminating unnecessary overhead due to distribution changes. As illustrated in Section 1 the essence of DAP is to avoid unnecessary executions of distribution assignments at runtime. Intuitively, a distribution assignment is *unnecessary*, 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. Hence, DAP relies on the combined effects of eliminating *partially dead* and *partially redundant* assignments.

This is important because both subproblems can optimally be solved as it was discussed in [7] and [8] presenting algorithms for *partially dead code elimination* (PDCE) and *partially redundant assignment elimination* (PRAE), respectively. Below, we are going to show how to enhance these algorithms being developed for a standard sequential program setting to the data-parallel setting, and how to combine them uniformly in order to arrive at a hierarchy of user-customized DAP-algorithms.

3.1 The Component Transformations of DAP: PDCE and PRAE

Like DAP, PDCE and PRAE consist conceptually of two elementary transformations each: *assignment sinkings* (AS) and *dead code eliminations* (DCE), and *assignment hoistings* (AH) and *redundant assignment eliminations* (RAE), respectively. Assignment sinkings (hoistings) move assignments as far as possible in the (opposite) direction of the control flow (i.e., while maintaining the program semantics). Intuitively, this places them in a context as *specific* (*general*) as possible, and maximizes the potential of dead (redundant) code, which subsequently is removed by dead (redundant) assignment elimination. The data flow analyses involved can almost straightforward be transferred from their classical counterparts, and are given in detail in [6]. Thus, we only briefly recall the algorithms of PDCE and PRAE, and go directly ahead to the optimality results applying to them.

According to [7, 8] the second-order effects induced by different assignment patterns are fully captured by repeatedly applying the elementary transformations of PDCE and PRAE until the program stabilizes. The algorithms for PDCE and PRAE are thus concisely and conveniently given by the following regular-expression like terms:

$$\mathsf{PDCE} \equiv (\mathsf{AS} + \mathsf{DCE})^{+} \quad \text{and} \quad \mathsf{PRAE} \equiv (\mathsf{AH} + \mathsf{RAE})^{+}$$

where AS (AH), and DCE (RAE) denote a single application of the assignment sinking (hoisting), and dead (redundant) assignment elimination procedure to all assignment patterns occurring in G. The programs resulting from PDCE or PRAE, respectively, are uniquely determined (up to irrelevant reorderings in basic blocks), i.e., they are independent of the specific sequence of the elementary transformations. Moreover, they are *best* (*optimal*), i.e., they are *better* than any other program in the sets of programs $\mathcal{G}_{pcde} =_{df} \{G' \mid G \vdash^{*}_{(\mathsf{AS+DCE})} G'\}$ and $\mathcal{G}_{prae} =_{df} \{G' \mid G \vdash^{*}_{(\mathsf{AH+RAE})} G'\}$

derivable from G by means of sequences of admissible assignment sinkings (hoistings) and dead (redundant) assignment eliminations. Central is the precise meaning of "better". It is defined as follows. A program G' is *better* than a program G'' iff for every assignment pattern the number of assignments executed on each path in G' is less or equal to that in G''. This is important because it directly implies that the programs resulting from PDCE and PRAE are best whatever the actual execution costs of the occurrences of a specific assignment pattern may be. As we are going to show next, for DAP it is necessary to take execution costs of assignment patterns into account.

3.2 Interdependences between PDCE and PRAE

As recalled above, the elementary transformations of PDCE or PRAE can be applied in any order without affecting the program finally resulting (cf. [7, 8]). Unfortunately, this property gets lost as soon as all four elementary transformations are interleaved as required for DAP. This is illustrated by the flow graphs of Figure 2. Applying PDCE first we arrive at the program of Figure 2(b), applying PRAE first we arrive at the program of Figure 2(c). 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 Figure 2(c) contains precisely one distribution assignment, there is a path through the fragment of Figure 2(b) being free of distribution assignments, and another one containing two.

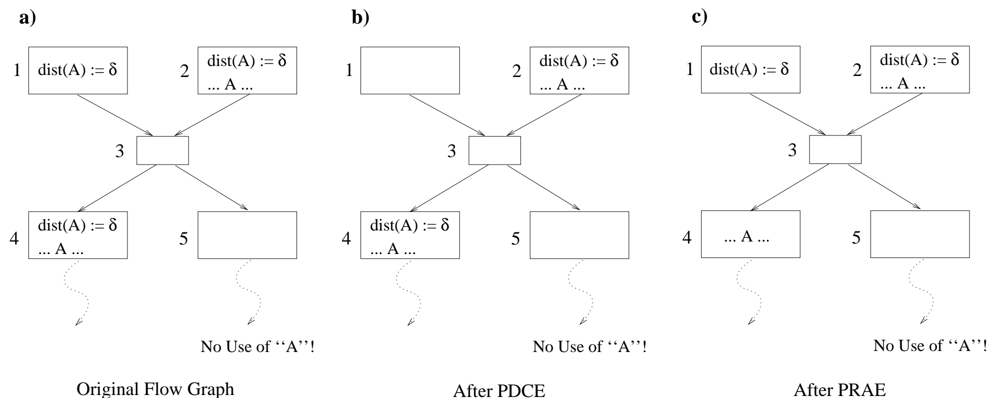


Figure 2: Interdependences between PDCE and PRAE.

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 like in this example "maximally transformed" programs are either identical (up to irrelevant reorderings in basic blocks) or incomparable wrt the order of Section 3.1 counting assignments for every assignment pattern separately. But programs may be considered incomparable for which an order according to the intuitive notion of "better" exists as demonstrated in the example of Figure 3. The programs of Figure 3(b) and 3(c) result from program 3(a) by beginning with PRAE and PDCE, respectively. On paths starting from node **1** program 3(c) contains less remappings than program 3(b). On path (3,4,5,7) the same remappings are required for both versions. On path (3,4,5,6), however, the distribution assignments dist(A) := BLOCK(i) and dist(B):= $\alpha_{\mathsf{A}}^{\mathsf{B}}$ are executed in 3(b), whereas in 3(c) the assignment dist(A) := BLOCK(i) is performed twice. Hence, both programs are incomparable wrt the relation "better" of Section 3.1. However, taking execution costs into account, both programs would perfectly be comparable, if the execution costs of dist(A) := BLOCK(i) and dist(B):= $\alpha_{\mathsf{A}}^{\mathsf{B}}$ coincide. Then, the program of 3(c) is strictly "better"

than that of 3(b); a fact, which is not reflected by the order of Section 3.1. Unfortunately, validating the equality of costs of different distribution assignment patterns is in general not trivial and requires a performance prediction evaluating program paths statically. Thus, it is worth noting that in program 3(b) 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 of 3(c) the second remapping is redundant. By means of *distribution assignment masking* (cf. Section 3.4) this unnecessary remapping can be suppressed at the price of a much cheaper runtime test implying that program 3(c) is "better" than program 3(b) without having to refer to the relative costs of dist(A) := BLOCK(i) and dist(B):= α_A^B. Note, however, that "better" relies now on a specific code generation strategy.

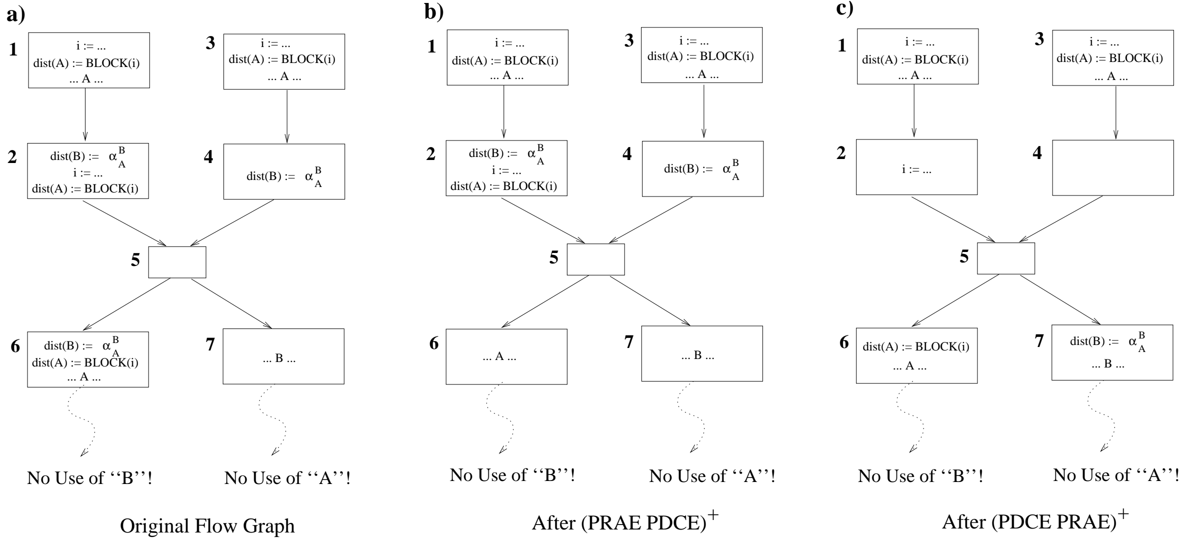


Figure 3: "Maximally" transformed programs are sometimes comparable.

In spite of these subtleties, we always have that the process of interleaving all four elementary transformations of PDCE and PRAE 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". Different transformation sequences, however, will usually come up with different locally best programs. In the following, we therefore concentrate on the question in which order to apply the elementary transformations to achieve fast stabilization. Obviously, AS- and AH-steps should always be interleaved with DCE- and RAE-steps (as AS and AH just reverse each other's effect). This implies interleaving of PDCE and PRAE which, in general, must be applied repeatedly in order to reach a stable state (see [6] for an example). In essence, this is a 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? Though in general any decision at this point is arbitrary, the following facts suggest starting with PDCE. First, the situation displayed in the example of Figure 1, where PDCE is a prerequisite for enabling PRAE, can be considered quite typical for data-parallel programs. Second, PRAE never creates partially dead assignments, but PDCE may create partially redundant ones. Thus, in spite of their conceptual duality, the effects and hence the interplay of PDCE and PRAE are not completely dual. Though not sufficient, this enlarges the chance that PDCE followed by PRAE already terminates with a locally best program as demonstrated in Section 1.

3.3 Pure DAP and Full DAP

Pure DAP focusses on the placement of distribution assignments. Before presenting this algorithm in detail, we remark that 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. Hence, decoupling assignment patterns are a major means for improving the performance of DAP. For distribution assignments this can be achieved particularly easily by a *preprocess* relying on reachability and constant propagation information. Note that this information is typically computed during compilation as it is required for several optimizations (see [6] for details). We remark, however, that decoupling assignment patterns is not a prerequisite of our approach, rather a means of improving efficiency. Next we present the algorithm of *Pure* DAP in detail. It is the iterated sequential composition of the algorithms for PDCE and PRAE applied to the set $\mathcal{D}$ of distribution assignment patterns occurring in G:[2]

$$\mathsf{DAP}_{pure} \equiv (\mathsf{PDCE}_{\mathcal{D}} \;\; \mathsf{PRAE}_{\mathcal{D}})^{+}$$

The Pure DAP-algorithm focusses on distribution assignment patterns. As a consequence the elimination of partially dead and redundant distribution assignments can get stuck by not considering the interdependences with ordinary assignments (cf. Figure 4 where the distribution assignment dist(A) := CYCLIC(k) is blocked by the ordinary assignment k at node 4). The Full DAP-algorithm thus differs from its counterpart for Pure DAP by taking all assignment patterns occurring in G into account:

$$\mathsf{DAP}_{full} \equiv (\mathsf{PDCE} \;\; \mathsf{PRAE})^{+}$$

Figure 4 illustrates the different power of both algorithms, for which the algorithm of Full DAP is unique to eliminate all partially dead and partially redundant distribution assignments.

3.4 Refined and Customized DAP-Variants: Enhancing Power and Efficiency

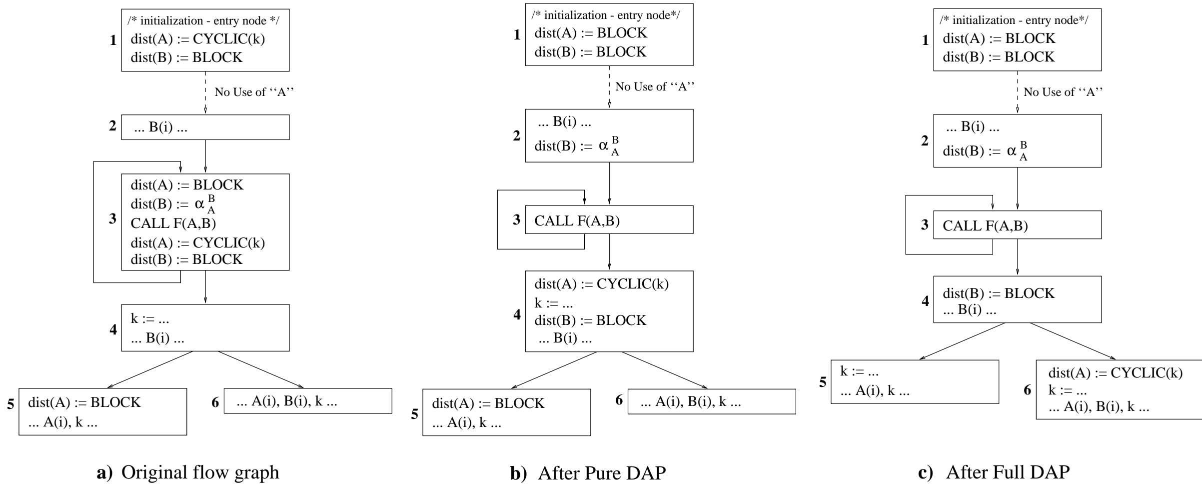
The algorithms of Pure DAP and Full DAP constitute the kernel of a hierarchy of DAP-algorithms of varying power and efficiency allowing customized DAP-variants according to a user's requirements. *Efficiency* for example, can simply be enhanced by limiting the number of iterations of the component transformations. The ratio underlying this heuristic to get algorithms being still reasonably effective is that distribution assignments are used in quite a restrictive manner only in practice. The extreme variant is here the one-step heuristic focussing on distribution assignments: $\mathsf{DAP}_{pure}^{one-step} \equiv (\mathsf{AS}_{\mathcal{D}} \;\; \mathsf{DCE}_{\mathcal{D}}) \;\; (\mathsf{AH}_{\mathcal{D}} \;\; \mathsf{RAE}_{\mathcal{D}})$. On the other hand, the transformational *power* of the DAP-algorithms can easily be enhanced by replacing the partial dead-code elimination procedure by the more powerful procedure for *partial faint-code elimination* (PFCE) (see [7] for details). Moreover, all DAP-algorithms can be combined with *distribution assignment masking*. This ensures that distribution assignments (at the price of a much cheaper runtime test) are executed at runtime only if they have a non-trivial effect. The flexibility the hierarchy offers a user for trading efficiency against power and vice versa for satisfying his requirements and preferences is illustrated in the following section.

4 Illustrating the Flexibility and Power of DAP

In this section we demonstrate the flexibility and the power of our approach by discussing the effects of Pure DAP and Full DAP on the example of Figure 4. It is complex enough to illustrate the essential features and differences of both algorithms, in particular, the power of Full DAP

[2] If the preprocess succeeds in decoupling all patterns of $\mathcal{D}$, $\mathsf{PDCE}_{\mathcal{D}}$ and $\mathsf{PRAE}_{\mathcal{D}}$ can equivalently be replaced by the more efficient algorithms specified by $(\mathsf{AS}_{\mathcal{D}} \;\; \mathsf{DCE}_{\mathcal{D}})$ and $(\mathsf{AH}_{\mathcal{D}} \;\; \mathsf{RAE}_{\mathcal{D}})$, respectively.

to resolve second-order effects caused by dependences between different assignment patterns completely. While Pure DAP handles dependences introduced by alignments only, Full DAP takes parameters in distribution specifications like in BLOCK(i) and CYCLIC(i) into account as well, as illustrated in Figure 4(a) and 4(c).



a) Original flow graph **b)** After Pure DAP **c)** After Full DAP

Figure 4: A complex example: Illustrating second-order effects.

Similar to our motivating example, subroutine call F in node **3** of Figure 4(a) causes four remappings inside the loop. Whereas Full DAP succeeds in removing all partially dead and partially redundant assignments as shown in Figure 4(c), there still remains one partially dead distribution assignment after applying Pure DAP as depicted by 4(b). Since Pure DAP does not take ordinary assignments into account, the distribution assignment dist(A) := CYCLIC(k) is blocked at node **4** by the assignment to k and, hence, is still partially dead due to node **5**.

The performance gain resulting from DAP may be tremendous, a fact which is not surprising in the light of average remapping times. For the highly efficient remapping routines of Ramaswamy et al. [12] the times are in average about 5 msec, 10 msec, and 32 msec on an Intel Paragon for remappings with different processor sets (from 8 to 16 processors), redistributions from (cyclic(3),block) to (cyclic,cyclic(5)), (cyclic(3),cyclic(7)) to (cyclic(5),cyclic), and (cyclic,$*$) to ($*$,cyclic) and arrays with sizes 100×100, 200×200, and 400×400, respectively.

5 Conclusions

Eliminating *partially dead* and *partially redundant* redistributions is of key importance to gain efficiency. Based on the recently developed algorithms for PDCE and PRAE of [7] and [8] working for standard sequential programs, we showed how to adapt and combine them to optimize data remappings in data-parallel languages. Second-order effects between PDCE and PRAE showing up by combining them required not only a refined optimality investigation, but also led to a hierarchy of algorithms for distribution assignment placement of varying power and efficiency offering customized solutions according to a user's requirements and preferences. This ranges from extremely efficient one-step heuristics to extremely powerful procedures resolving all second-order effects between different assignment patterns like the enhanced Full DAP-Algorithm. Currently, we are investigating an interprocedural extension of our approach along the lines of [9], and how it compares to other interprocedural algorithms. An implementation of our approach within the VFCS system [1] is in progress.

References

[1] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B.M. Chapman, M. Egg, T. Fahringer, J. Hulman, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H.P. Zima. *Vienna Fortran Compilation System - Version 1.2 - User's Guide*. Institute for Software Technology and Parallel Systems, University of Vienna, Vienna, February 1996.

[2] F. Coelho and C. Ancourt. Optimal compilation of HPF remappings. *Journal of Parallel and Distributed Computing*, 38(2):229–236, November 1996.

[3] High Performance Fortran Forum. High Performance Fortran language specification version 2.0. Technical report, Rice University, Houston,TX, January 1997. Available via HPFF home page: http://www.crpc.rice.edu/HPFF.

[4] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. FORTRAN D language specification. Technical report, Rice University, Houston,TX, January 1992.

[5] M. W. Hall, S. Hirandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proc. of Supercomputing '92*, pages 522–534, Minneapolis, November 1992.

[6] J. Knoop and E. Mehofer. Distribution assignment placement: A new aggressive approach for optimizing redistribution costs. Technical Report TR 97-6, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, 1997.

[7] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 147–158, Orlando, Florida, June 1994.

[8] J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 233–245, La Jolla,CA, June 1995.

[9] J. Knoop, O. Rüthing, and B. Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Languages*, 4(4):211–246, 1996.

[10] E. Mehofer and H. Zima. Distribution assignment placement. Technical Report TR-96-5, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, 1996.

[11] D.J. Palermo, E.W. Hodges, and P. Banerjee. Interprocedural array redistribution data-flow analysis. In *Proc. of the 9th Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[12] S. Ramaswamy, B. Simons, and P. Banerjee. Optimizations for efficient array redistribution on distributed memory multicomputers. *Journal of Parallel and Distributed Computing*, 38(2):217–228, November 1996.

[13] 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, March 1992.