The Integration of ParaScope and Lambda

Donna Bergmark and David Presberg
Cornell Theory Center *

August, 1993
Revised: May, 1994

Abstract

We have been experimenting with combining three powerful language tools for large, scientific, parallel Fortran codes. One tool is ParaScope, a programming environment; another tool is the Lambda Toolkit, a collection of routines for performing loop transformations using invertible matrices; the third is FORGE 90, a collection of tools for parallelizing Fortran programs. Initial success with incorporating the Lambda Toolkit into ParaScope led us to undertake the work leading to a new program preparation strategy, in which one first uses a modified ParaScope to perform Data Access Normalization, then uses FORGE 90 to produce a parallel program for a distributed memory platform.

We describe the details of this strategy and present some performance results for the IBM SP1. We conclude that the combination of ParaScope and the Lambda Toolkit (called "ped-Lambda") is a useful transformation tool.

Keywords: tools, tool combinations, data locality, APR FORGE 90 DMP, HPF, data access normalization, ParaScope, Lambda Toolkit, SP1, Fortran, parallel

CTC94TR180
6/94

*This work was partially supported by NSF New Technologies project ASC 9201498, entitled "The Evaluation and Deployment of ParaScope".

Contents

1 Introduction

It is now possible to produce parallel programs for high performance computers using only parallel programming tools. Hand coding of parallel constructs or message passing calls is unnecessary.

In an earlier report [4], we described how we approached the integration of two very powerful software systems, ParaScope and the Lambda Toolkit. We discussed the extensibility of ParaScope, the generality of the Lambda Toolkit, and things we learned from the integration exercise.

In the work reported here, we describe a fully integrated system consisting of the main interactive tool in the ParaScope Toolset and a new release of the Lambda Toolkit. Using this new combination tool and the latest version of FORGE 90, we have transformed user source codes and achieved parallel speedup on the SP1. Those results are also reported in this paper.

2 Software Components

This section briefly describes the software used in the integration of ParaScope and the Lambda Toolkit. It also describes the FORGE 90 Distributed Memory Parallelizer (DMP) and High Performance Fortran precompiler (xHPF).

2.1 ParaScope

ParaScope [5] is the outgrowth of Rice University research into programming environments. The ParaScope Toolset is a collection of 12 X-based interactive tools and Unix filters organized around Fortran source-to-source transformation techniques. Each tool is invoked from a Unix command line. One intent of the collection is to provide prototype tools for modifying serial source codes for parallel execution.

The ParaScope infrastructure contains a full compiler front-end for Fortran 77, a dependence analysis subsystem, interprocedural analysis, and a Fortran source text regenerator. ParaScope is distributed with a full complement of source and development components for each system on which it is hosted. The source is a mixture of C and C++ that is acceptable to the versions of the `g++` and `gcc` compilers supplied with the release. Extensive use is made of `gnumake`. Libraries of utility routines support X-based user interfaces, transformations of the internal representation of a Fortran source module, and general tool housekeeping. Existing interactive tools in the ParaScope Toolset permit syntax-driven structure editing or source text editing. In particular, the PEDX editor highlights views of dependence information, and has a menu of non-trivial source transformations, sensitive to the content of loop nests the user selects for transforming.

Rice University provided "Release 1" of the ParaScope Toolset in April 1993. There have been a few bug fixes of particular modules since then, but no new overall release. Approximately 20 sites have worked with ParaScope, some developing additions or modifications to the tools. The developers are now engaged in creating a replacement called the "D System" (e.g., [13]).

2.2 Lambda Toolkit

The Lambda Toolkit [15] is a library of C routines that rewrites a nest of loops according to transformations that can be specified as integer matrices. It is equally applicable to DO loops in Fortran and DO-style **for** loops in C. The Lambda Toolkit is based on the non-singular matrix transformation theory described in [18]. The main thrust of this theory is as follows.

One uses integer matrices to transform loops by treating the loop indices as a vector `<i, j, ...>` and the matrix as a linear transformation on that vector.[1] Each reference to an original loop index is replaced by a linear combination of new loop indices from the *transformed* iteration space. Naturally, the matrix must be full-rank (i.e., square in the number of loop indices or loop nest depth).

If the transformation matrix is invertible then there is a one-to-one mapping between the original iteration space and the transformed iteration space. This observation is important because if one knows a better shape for the iteration space — say to increase opportunities for parallelism — the loop bounds are transformed to generate the improved iteration space. Then one simply replaces all the subscript expressions within the loop by the application of the inverse of the transformation that provides the better iteration space. An example is in Appendix A.

The Lambda Toolkit incorporates this theory in an efficient implementation that generates loop bounds and step sizes directly, given a loop nest and a transformation matrix. (Previous methods inserted conditionals into the loop body in order to skip illegal index values.) To facilitate rewriting loop nest bounds, the Lambda Toolkit contains a routine which, given a structure representing initial, final, and step expressions of a nest of DO loops, and a non-singular matrix, will return a similar structure completely representing the transformed loop bounds. Also, the Lambda Toolkit contains a routine which, given a linear combination of the original index variables represented as a vector of coefficients, will return the equivalent vector in the transformed space. This allows you to replace all expressions in the old index space with equivalent expressions in the new space.

[1] Nomenclature: The Greek letter Lambda (Λ) is often used as a mnemonic for linear transformations. Here they are one-to-one mappings between spaces that include skewing, scaling, general permutation of coordinates, and coordinate reversal.

The toolkit is also easy to use because incomplete transformations can be specified. If you give the Lambda Toolkit a transformation that is not square, it will automatically fill the matrix so that existing loop dependences are respected and the resulting matrix is still invertible. For example, if you wanted to reverse the iteration order of the outer loop but did not care what happened to the inner loop, you could code just the reversal and the algorithm would fill out the matrix, keeping it non-singular while preserving the dependences present in the original loop nest; lexicographically positive dependence vectors in the original loop nest remain positive in the transformed loop nest.

The Lambda Toolkit is available as public domain software from either the Cornell University Computer Science Department or from the University of Rochester Computer Science Department[2]. Program access to the Lambda Toolkit is given by a library, `libLambda.a` and an include file. The Lambda Toolkit is easily built on SUN workstations or on IBM RS/6000 AIX workstations with C or C++ compilers. The toolkit has a simple interface based on ordinary C structures, which are independent of any intermediate representation the surrounding software system might have.

2.3 FORGE 90

FORGE 90 is one of a number of commercial products for analyzing, improving, and parallelizing Fortran source code, supplied by Applied Parallel Research (APR). Currently, one uses the main interactive tool, FORGE 90 Baseline [22], to gain insight into the structure of, to instrument, and to aid in maintenance of source collections of any size and complexity. From Baseline, the user can engage the FORGE 90 Distributed Memory Parallelizer (DMP) [24] to generate an SPMD parallel code focussed on data parallel execution of computation-heavy loops. Alternatively, one may capture these decisions as HPF [12] directives and other directives in the source program and process it through the FORGE 90 HPF preprocessor (xHPF) [23] to obtain an SPMD code.

As we explained in our earlier technical report, FORGE itself does not reorganize a Fortran program to enhance parallelism. It takes the source code given to it, and converts it to parallel form if it is legal to do so. The output of FORGE 90 DMP is a Fortran 77 program with inserted APR message passing subroutine calls which can run on top of popular communications libraries such as PVM [7, 26], EXPRESS [21], Linda [6], or hardware-vendor-supplied communications libraries such as that for the Intel multiprocessors [11] or the IBM 9076 Scalable POWERparallel Systems (SP) [25].

[2]At Cornell, the anonymous ftp server is `ftp.cs.cornell.edu` and the sources, documentation, and some examples are in the file `pub/TyphoonCompiler/lambda-25Sep93.tar.gz` At the University of Rochester, the anonymous ftp server is `cs.rochester.edu` and the sources, documentation, and examples are in `pub/wei/lambda-25Sep93.tar.gz` The developer, Wei Li, can be contacted at `wei@cs.rochester.edu`

Subsequent to our previous work [4], APR upgraded the communication support for DMP to include the optimized hardware-switch-using native communication libraries of the SP1, "MPL" and "MPLp" [10]. Also, APR shipped the first subset HPF preprocessor in the marketplace, xHPF. FORGE DMP shares much of its analysis facility and rewriting technology with the xHPF tool.[3]

3 *Data Access Normalization*

When parallelizing programs for distributed memory machines, a common technique is to allocate portions of arrays to the various memories and then assign loop iterations to individual machines such that most memory references are local. Accessing data not on that machine is a remote access. These must be kept to a minimum in order to achieve good performance.

Data access normalization as described by Li and Pingali [16, 17] is a transformation that reorganizes the iteration space of a loop nest in order to reduce the number of remote accesses required to fetch data. It also allows individual fetches to be combined into a single larger transfer whenever possible. Figure 1 shows an example of a loop before and after data access normalization. The bounds have become more complicated, but the array references have been simplified and the transformed loop is readily parallelized. In addition, a block of data can be fetched all at once, rather than an element at a time. On a message passing system with high latency, this can make a large difference in run time. The Lambda Toolkit contains many functions to support data access normalization; indeed, this was one of the first demonstrations of the toolkit's use.

Data access normalization is based on analyzing array subscript expressions found in the loop body with respect to how the arrays will be decomposed across distributed memories. If one maps the most frequently used subscript expressions to single new loop index variables, and if those index variables are in distributed (i.e., parallel) loops, then that data is local according to the *owner computes* rule. This rule is frequently used in large parallelizing systems like FORGE DMP (see Figure 2).

Normalizing subscripts in this way would be helpful in optimizing programs for workstation clusters and the SP series, both of which are distributed memory systems. However, it is sometimes difficult to rewrite loops by hand in order to achieve data access normalization. Our goal, therefore, was to build a tool by extending ParaScope to include the techniques described here. The extended tool is called pedLambda. We would then use FORGE 90 DMP or xHPF to generate the appropriate message-passing program. See Section 6 for how we used the new version of FORGE with pedLambda.

[3] At the end of the project, we were using the FORGE 90 Version 8.9 release of APR's tools.

```
subroutine ala_fig1( A,B,N1,jb,N2 )

       real A(100, 100), B(100, 100)
       integer N1, jb, N2

       integer i, j, k

       DO i = 1, N1
          DO j = i+1, i+jb
             DO k = 1, N2


             B(i, j-i) = B(i, j-i) + A(i, j+k)

             ENDDO
          ENDDO
       ENDDO

       end

(a) Un-normalized loop
```

```
subroutine ala_fig1_uvw(a, b, n1, jb, n2)

        real a(100, 100), b(100, 100)
        integer n1, jb, n2

        integer u, v, w

        do u = 1, jb
          do v = u + 2, u + n1 + n2
            do w = max( 1,  v-u-n2),
     .             min( n1, v-u-1)

              b(w, u) = b(w, u) + a(w, v)

            enddo
          enddo
        enddo

      end

(b) Normalized loop
```

Figure 1: Case related to Figure 1 in Li and Pingali [16]. Shown is a loop nest (a) before and (b) after data access normalization. (We have corrected the generated loop bounds of the innermost loop to correspond to Lambda Toolkit correct results.) Assume that the outermost loops are distributed but that the innermost loop is executed by a single processor for any given combination of the outer indexes. Assume also that A and B are to be decomposed by columns. The idea behind data access normalization is that it allows block data transfer of matrix `A`. In (a), each reference to A(i, j+k) requires a remote access. In (b), the process can receive an entire column of A before beginning the `w` loop.

4 *FORGE, reprise*

The FORGE DMP tool increased in capability from the version we used for our initial report so that it became more sensitive to dependences carried through loop bounds expressions. At the same time, its analysis determined more cases in which optimal *scalar* and *vector reductions* could be used. Comparison of the example in our initial report against FORGE DMP output on current Lambda Toolkit transformation results shows this increased capability. (Note in Figure 2 the `Perform_recv...` and the `Perform_send....`)

As a test of FORGE DMP's code generation capabilities, we edited the Lambda transformed

```
Viewing Parallelization of ALA_FIG1_UVW/DO/ENDDO (2) V

              subroutine ala_fig1_uvw(a, b, n1, jb, n2)

                real a(100, 100), b(100, 100)
                integer n1, jb, n2

                integer u, v, w

   ---> partition A (*,SHRUNKCYCLIC) move
                do u = 1, jb
   ---> Distribute the loop on A<:, 2+u~1>
   ---> Use of  A<:, 2+u~1>
   ---> Use of  B<:, u>
   ---> Postloop communication of  B<:, u>  replicated
                  do v = u + 2, u + n1 + n2
   ---> Perform_recv on  B<:, u>
                    do w = max( 1,  v-u-n2),
             =               min( n1, v-u-1)

                      b(w, u) = b(w, u) + a(w, v)

                    enddo
   ---> Perform_send on  B<:, u>
                  enddo
                enddo

              end
```

Figure 2: Demonstration that FORGE is able to generate block data transfers, given the appropriate loop nest as input. In this case, we asked for array A to be decomposed on the second index (that is, by column) and the middle loop to be distributed. This causes block data transfer of `B`, a column at a time (see `Postloop communication of B<:, u>`). Each node will have all the rows of A in the appropriate columns for index v, i.e. $u+2$ through $u+N1+N2$ (see `Use of A<:, 2+u~1>`), assuming the owner computes rule. Similarly, B — which is replicated everywhere for this example — is temporarily owned (at least in part) by the processor executing the innermost loop.

output to (over)simplify the loop bounds on the inner `do w` loop. Figure 3 shows this case. The simpler bounds allow FORGE to recognize the vector reduction on column `u`.

```
Viewing Parallelization of ALA_FIG1_UVW_SIMPL/DO/ENDDO (2) V

     1:         subroutine ala_fig1_uvw_simpl(a, b, n1, jb, n2)
     2:
     3:           real a(100, 100), b(100, 100)
     4:           integer n1, jb, n2
     5:
     6:           integer u, v, w
     7:
   ---> partition A (*,SHRUNKCYCLIC) move
     9:           do u = 1, jb
   ---> Distribute the loop on A<:, 2+u~1>
   ---> Reduction function ADD on  B<1+u:n1+u, u>
   ---> Use of  A<1+u:n1+u, 2+u~1>
    13:             do v = u + 2, u + n1 + n2
    14:               do w = u+1, u+n1
    16:
    17:                 b(w, u) = b(w, u) + a(w, v)
    18:
    19:               enddo
    20:             enddo
    21:           enddo
    22:
    23:         end
```

Figure 3: Demonstration of FORGE DMP results for edited `do w ...` bounds that are not dependent on the distributed loop index, `v`. The same column decomposition of A is used, and the `do v ...` loop is selected for parallel distribution. Note the simpler addition reduction over the rows in the `u` column of B as compared to the within-loop communication shown in Figure 2.

5 *The Integration of ParaScope and Lambda*

To lay the groundwork for understanding the new tool to normalize Fortran programs interactively, we recap the non-interactive processor, PALA, that was the precursor for the work done here. We then describe the new interactive processor, PEDLAMBDA, that integrates ParaScope's analysis and transformation framework with data access normalization and the Lambda Toolkit.

5.1 *The Final* PALA *Non-interactive Processor*

We described the design evolution of PALA, our proof of concept, in [4]. The implementation also evolved after that publication. With more confidence in the capabilities of the emerging batch tool, and with an upgrade of the Lambda Toolkit, we found it easy to remove some of the data reformatting steps inherent in the old pala_glue layer[4]. The ParaScope side was converted to use new Lambda Toolkit interface structures, and of Lambda Toolkit functions such as la_bounds and la_nest were invoked directly in PALA. The new Lambda Toolkit expanded in capability to handle non-unit-stride loops, so the norm function was no longer needed. Also, at this point, pala_glue was no longer needed, and the tgen function was called directly by PALA.

Note that the PALA tool was capable of applying only data access normalization to only one loop nest. The ability to data access normalize full Fortran programs came with PEDLAMBDA.

Figure 4 shows the data flow in PALA. Actions that derive or modify data are depicted in numbered rectangles, the data in labeled ovals. Briefly, the actions are:

1 Read source text of module, parse it, and construct an Abstract Syntax Tree (AST). Other data structures, such as statement text and dependence information are associated with the AST.

2 Traverse the AST, find all the DO statements in the loop nest, make tables of the DO loop bounds expressions (bounds) and the DO index variables (DO vars).

3 Within the body of the loop nest, traverse all the statements, find array references, traverse their lists of subscript expressions. Each subscript expression either is categorized as a linear combination of the DO variables, or other. For the linear subscript expressions (linear ss), tabulate the coefficients of the DO variables, the constant term, and the position in the AST. For the subscript expressions that aren't linear (nonlinear ss), record only the position in the AST.

4 Traverse the nested AST structure of the DO statements in the loop nest. Lists of dependence vectors are associated with the DO loop that carries the dependences. All dependence vectors are accumulated on a Lambda Toolkit structure (dep). Dependence vectors are translated from a ParaScope representation to a Lambda Toolkit representation.

5 Generate a transformation matrix (xform) by applying the data access normalization algorithm (see section 3) to the subscripts, the dependence matrix, and data distribution from the file ./distribution. This step invokes several dependence manipulation routines from the toolkit. It calls la_complete() to fill out the transformation matrix. (In PALA, this is function tgen.)

6 Invoke the toolkit function la_nest on the loop bounds (in Lambda Toolkit form) and the transformation matrix from step **5** to produce the transformed loop bounds (xform bounds),

[4] One reason for the pala_glue layer was a mismatch in the versions of the Lambda Toolkit interface used on the ParaScope side (hypothesized newer interface) and on the tgen side (older actual interface). After an updated Lambda Toolkit was released, both sides adopted the new interface.

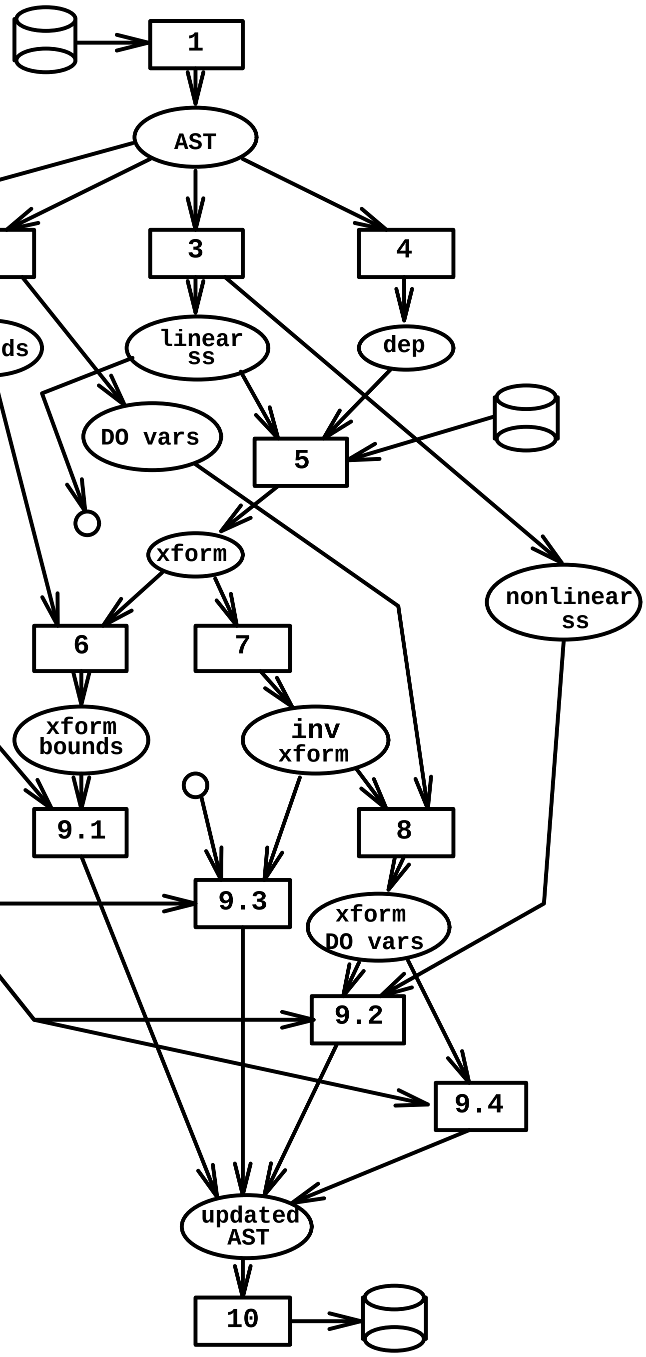

Figure 4: Data flow diagram of PALA batch-mode tool.

still in Lambda Toolkit representation.

7 Invert the transformation matrix (invoke toolkit function `la_matInv`), producing the inverted transformation matrix (`inv xform`).

8 Each row of the inverse transformation matrix specifies the linear combination of new loop indices to be used to replace an original loop index (DO variable) where that original variable appears in the body of the loop (see section 2.2). This action prepares for that future replacement by constructing and tabulating the AST structures of the linear expression replacements (`xform DO vars`). (These AST "tree-lets" are maintained outside the main AST structure, but are represented in ParaScope form.)

9.1 Traverse the DO statements in the AST, replacing the old bounds and increment (step) expressions with the new, now converted to ParaScope form. Note that the new expressions often involve integer division and the use of Fortran intrinsic functions $max()$, and $min()$, and the $ceil()$ property of integer division: all bounds expressions are guaranteed by the Lambda transforms to produce integer iteration spaces. (The DO statements use their original DO variable identifiers, from outer to inner — these are unchanged in the DO statements themselves.)

9.2 Traverse the list of AST locations where the subscript expressions are not linear. For each such subscript expression, walk its AST replacing each instance of a DO variable with the corresponding tree-let.

9.3 Traverse the list of linear subscript expressions. For each, transform the linear expression (still in a Lambda Toolkit form) by applying the inverse transformation matrix to the coefficients of the linear subscript (invoke toolkit function `la_vector`). Simplify the resulting linear expressions (using the Lambda Toolkit `la_gcd` function), and construct a new AST tree-let for the replacement expression. Replace the old subscript expression with the new in the AST, at its tabulated location.

9.4 Walk the AST body of the loop nest, but don't process the children of any array reference. (They have already been updated by steps 9.2 and 9.3.) Replace remaining references to original DO variables with their transformed expression, all in ParaScope form.

10 Unparse the AST and produce text of the transformed source module to `stdout`.

In PALA, there was an attempt to construct locally, relatively clean expressions (note the use of `la_gcd()` in step 9.3). However, there was no mechanism immediately available to do algebraic simplification, and so the code generated by PALA was fairly ugly, though correct. As mentioned above, PALA's input was limited to a Fortran module containing a single loop nest with constant bounds.

5.2 *Interactive* pedLambda

The new tool, pedLambda, is an extension of the ParaScope Toolset's pedX interactive Fortran editor. pedX was designed to enable a Fortran programmer to transform the source code of a Fortran module, in various ways, to enhance parallelism. We will describe some aspects of the pedLambda extension here is order to motivate implementation details discussed in Section 5.3.

5.2.1 *User Interface*

In Appendix B of our previous report we showed the full list of transformations that pedX can perform. To those we added the "lambda-based" transforms as a selectable menu item (see Figure 5). The unextended pedX has only the first three categories of transformations in its `Transformations` menu.

The new `lambda-based` menu lets the user select a single loop nest or all the loop nests in a module for conversion. It also allows the selection of one of a number of transformation "Preferences", all of which are implemented with Lambda Toolkit technology (see Figure 6).

When the user clicks the mouse on either the `selected loop nest` or the `all loop nests` menu choices, the Lambda-based transformation is applied to the Fortran source code. The upper pane of the pedLambda window then shows the new source code. For instance, Figure 7 shows the Data Access Normalization transformation applied to the selected loop nest, with the second subscript of each of the `A` and `B` arrays indicated for parallel distribution. Since the result is still available for further edits with the entire menu of ParaScope transformations, the user can select other loop nests for Data Access Normalization, fuse loops with identical bounds, "distribute" loops to form "tight" loop nest structures (anticipating more Lambda-based transformations), or any other ParaScope-supported transformation.

All prospective transformations or subsidiary menu choices can be cancelled by clicking on the **X**-ed box in the upper left corner of each ParaScope dialog or submenu. Finally, when the source code has been improved to the user's liking, it can be rewritten as compilable text to a file, through the use of the various `save...` dialogs in the `file` menu, or during a final "failsafe" dialog that ParaScope presents as the user exits the tool.

5.2.2 *Handling Data Decomposition Information*

For the Lambda transforms to accomplish data access normalization, there needs to be information supplied to the system about which arrays should be partitioned, and on what axis. Directives embedded in the Fortran source text would be the ideal way to supply this information. Since

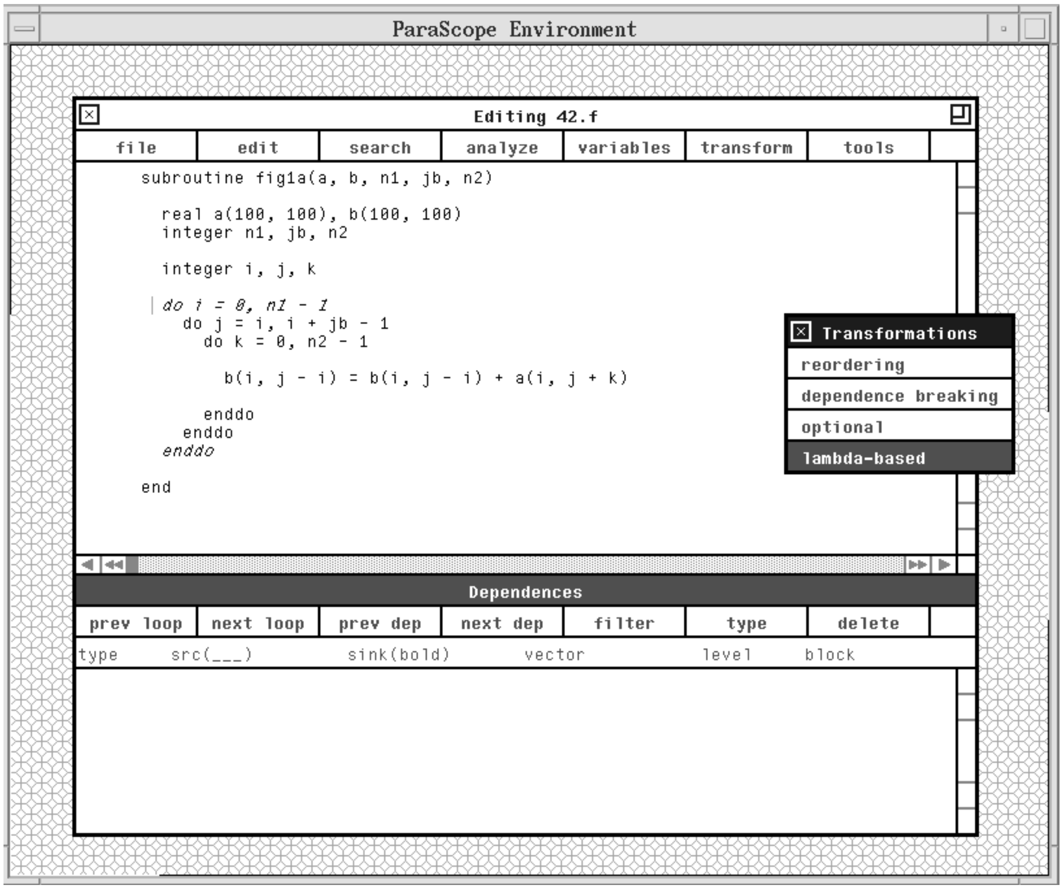

Figure 5: A ParaScope X window instance running PEDLAMBDA with the `transform` tear-off menu selected, and the mouse ready to select the `lambda-based` transformations. Note how the ParaScope selection focus has been applied to the outermost loop, `do i ...`, which now has a vertical selection-stroke to its left and is presented in a slanted font.

directives have the syntax of Fortran comments, the code is still a "sequential" code for debugging purposes. Also, with industry-accepted standard directives, such as the HPF data decomposition directives, the code is portable at the source level.

The ParaScope Toolset does include the Fortran-77D MIMD Compiler [8, 27], and PEDX can invoke it from the `tools` menu. This tool accepts Fortran-D directives to drive automatic data decomposition and parallelization transformations. (Fortran-D is a precursor of HPF.) However, as with much of the rest of ParaScope, documentation of the exact source language accepted by this tool, particularly the syntax of its directives, was nonexistent. Also, the implementer graduated from Rice University and left CRPC just as the final stages of our project were starting. Rather than

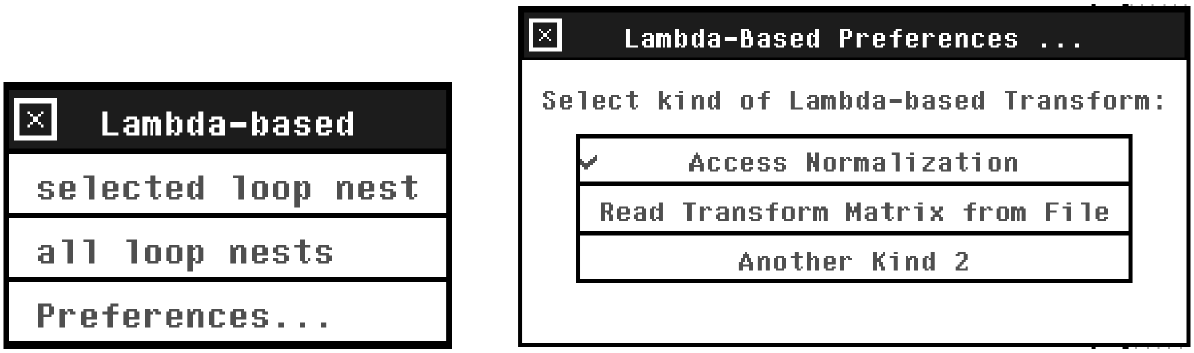

Figure 6: The menu on the left shows the choices for transforming either one selected loop nest, or all loop nests of a module. If no loop nest has been selected, the first menu choice will be disabled and will appear dimmed. The menu on the right is presented when the user selects `Preferences...`. It shows the alternative choices for how the transformation will be done. The first, checked-marked, selection is data access normalization. The second choice uses a transformation matrix read by the tool from a text file. The third choice is currently a place-holder for expansion. The Preferences choice persists from one use of the Lambda Based Transformations to another, and defaults to Data Access Normalization.

---

using uncertain Fortran-D processor mechanisms, therefore, we decided to retain PALA's convention of reading decomposition directives from a text file (step `5` in Figure 4). However, for PEDLAMBDA, we decided to improve the file's ease of use.

The form of that original file was a sequence of integers, `1` or `0`, indicating whether or not a particular array subscript was distributed or not, respectively. There needed to be exactly as many `1`'s or `0`'s as there were subscript expressions in the Fortran loop nest being processed, and they had to be present in exactly the order in which the PALA tool would encounter each subscript expression in its AST tree walk. The new file, now named `./decomposition`, has the following information just once for any subscript position for each named array:

- the text of the array name (in lower case letters because of how ParaScope represents identifiers in the AST)
- an integer denoting the ordinal position of an array subscript
- this position's distribution code: 0=not, 1=BLOCK, 2=CYCLIC, etc.

Thus the `./decomposition` file is much easier for the user to construct than the `./distribution` file had been. Also, the information in the new file format resembles quite closely information that might be easily processed to or from HPF directives. And since the information is linked to array name rather than to a particular reference within a particular loop nest, the file can be used for the entire program.

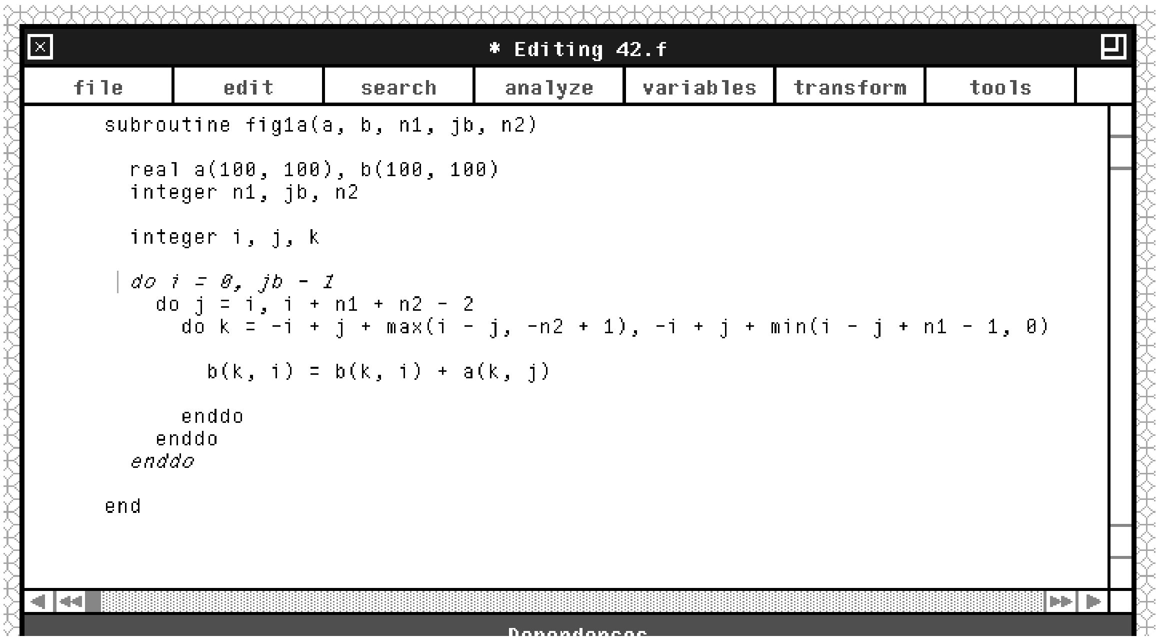

Figure 7: The selected loop nest has been transformed by Data Access Normalization, given that the second array subscripts are to be distributed. Note how the former `(i, j - i)` subscript expression-pair (Figure 5) has been "rotated" to `(k, i)` in the new loop nest. The compensating transformation of the loop bounds assures that the same elements of `A` and `B` are fetched and stored, as in the original loop nest.

5.2.3 *A Data Access Normalization Strategy Using* PEDLAMBDA

We now describe how PEDLAMBDA can be used to transform a serial Fortran program using data access normalization techniques. First, prepare a `./decomposition` file, possibly transcribing it from Fortran-D or HPF directives in the source program. Then, invoke PEDLAMBDA on the program. Detect and remove as many data dependences as possible for loops that will be parallelized, using "traditional" ParaScope functions.[5] Leave reductions alone, though, since FORGE now automatically handles these appropriately (see Section 4).

Lambda transformations currently apply only to tight loop nests, so use PEDLAMBDA to distribute any imperfect loop nests. This creates additional single loops, but leaves tightly nested loops behind (see Almasi and Gottlieb [1] for details on loop distribution).

Then use PEDLAMBDA's data access normalization button on all selected loop nests. This may cause inversion, loop reversal, loop scaling, and/or loop skewing. After transforming as many loop

[5] Loop alignment is one example of a loop transformation that will not be handled by data access normalization.

nests as desired, one can fuse any adjacent loop nests that now have the same outermost bounds. (See [3] for a detailed description of this overall procedure.) The result is a code that can be parallelized for distributed memory machines.

5.2.4 Additional Transformation Strategies

Interactive PEDLAMBDA could be extended in many ways. We have been concentrating on data access normalization, but other transformation strategies are possible. The simplest way to specify a different transformation is to read the transformation matrix in from a file, so we early on added that as a second kind of "Lambda-based transform" (see Figure 6). But other predefined transformations could be incorporated as well, such as wavefront normalization.

Consider that it is often the case that the parallelism in a program lies in sweeping successive "wavefronts" across the iteration space, rather than doing the parallelism rectangularly. For example, in the following loop (taken from Banerjee [2]), neither the inner nor the outer loop can be executed in parallel:

```
do i = 5,100
  do j = 5,100
    x(i,j) = x(i,j-1) + x(i-2,j+3) + x(i-3, j+7)
  enddo
enddo
```

However, this loop nest can be converted to a form that can be run in parallel. For example, if

$$T = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix}$$

is applied to the above loop nest, the loops are interchanged and skewed. In the resulting loop nest, the inner loop carries no dependences and can be run in parallel. Here is the transformed program:

```
   do i = 20, 400
     do j = max(5, (i-98)/3 ), min(100, (i-5)/3)
        x(j, i-3*j) = x(j, i-3*j-1) + x(j-2, i-3*j+3) +
+             x(j-3, i- 3*j + 7)
     enddo
   enddo
```

It has 9216 iterations, just as the original loop does, but now the loop carried dependences are all on the outer loop. A future project might be to use the Lambda Toolkit to implement a wavefront transformation generator.

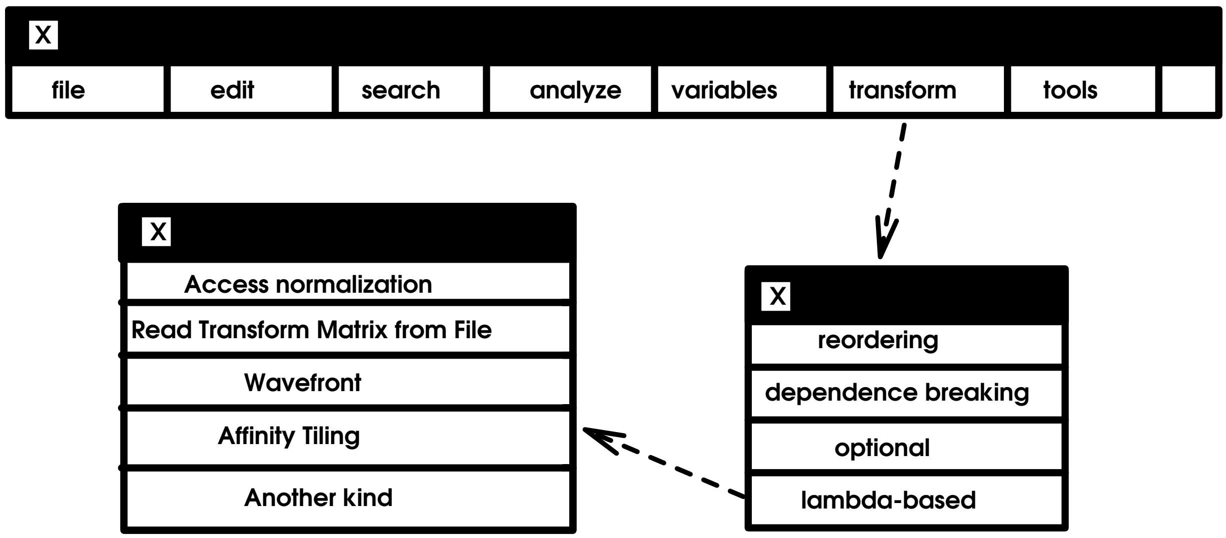

Figure 8: What the transform menu might look like with additional transformations in addition to the data access normalization transformation.

See [14] for yet another transformation for cache optimization, called "height reduction". The affinity tiling algorithm described in that report, for example, could be added to the Lambda-based transformation menu as well. Figure 8 shows what the new transformation menu might look like.

5.3 System Evolution

Embedding the transformation-producing code of PALA in PEDX to produce the extended tool, PED-LAMBDA, proved easier than we first feared. Fortunately, we had an example of a similar extension of PEDX to study. Early in 1993, Kaus-Peter Wiedemann *et. al.*, of Siemens AG, Munich, Germany, extended a version of PEDX to include what he called "Autopar", a shared-memory transformation heuristic based on PEDX transforms. They extended the `Transformations` menu, adding an item and submenus to invoke `automatic parallelization`. By comparing their modified source code against the base ParaScope sources used in PEDX, we were able to find most of the places where we had to change user-interface-handling code and transformation-driving source code. This, plus an August 1993 walk-through by Klaus-Peter during a visit to Cornell, proved invaluable.

Our approach was to move most of `pala.C` into a lower-level module of PEDX. We decided to leave the system-building mechanism for ParaScope Toolset parts largely unmodified. ParaScope tools are built from a collection of libraries, themselves built from numerous C and C++ source files, organized into a large source tree structure. Lacking maintenance documentation for this structure we often mimicked existing practice. For instance, the developers of ParaScope provided for development "sandbox" filespace: separate source tree structures similar to the base structure. Thus, rather than create an entirely new tool executable in the base ParaScope source tree, we started a "sandbox" version of PEDX and called it "pedLambda". It contains only a few added or modified files which override the original source tree, as shown in Figure 9.

The user interface code, `ped_cp`, is split into two subtrees, `dp` for handling X-based dialogs, and `pt` for carrying out user-selected actions. We added one new module to each subtree: `lambda_d.[ch]` to support the `Preferences...` dialog, and `pedLambda.[Ch]` to do Lambda transformations.

```
       $MY_SRC
         / \
        /   \
       /     \
     ...    libs
             / \
            /   \
           /     \
         ...    ped_cp
                /   \+----------+
               /     \           \
              /       \           \
             /        pt           \
            /     -----------       \
           /      Makefile         ped_include
         dp       trans.c         -------------
------------      pedLambda.C     ped_dialogs.h
Makefile          pt_util.c       lambda_d.h
trans_menu.c                      pedLambda.h
lambda_d.c                        pt_util.h
```

Figure 9: Partial source code tree in "sandbox" version of PEDX. `$MY_SRC` is a UNIX environment variable that anchors the ParaScope development system at the root of the developer's sandbox source tree. This source tree overrides identical files rooted at the path defined by the environment variable `$RN_SRC`.

The contents of the new `pedLambda.C` file is mostly the old `pala.C` file. In terms of the data flow chart of Figure 4, the omitted code corresponds to action boxes `1` and `10`, involving

Fortran source-file reading, parsing, analyzing, unparsing, and text output. The remainder became `pt_pedLambda()`, called with an AST already constructed, and a single loop nest already selected, but otherwise executing actions `2` through `9.4` just as PALA had done.

To invoke `pt_pedLambda()` we added code to both the `dp` and `pt` sides of the user interface subtree. On the `dp` side, we mimicked the "Autopar" additions to menu-creation and event-management code. On the `pt` side we added some "thin glue" over the top of `pt_pedLambda()` to accommodate PEDX conventions. In particular, we added `makePaLambda()` in `trans.c`, implementing the PEDX protocol for examining and potentially transforming one selected loop nest. The design of that function was taken directly from a similar example in the "Autopar" modifications.

The menu selection, `all loop nests`, in Figure 6 invokes `makePaLambdaAll()`, also in `trans.c`. This function traverses the AST from the top; whenever it finds a DO statement, it invokes `makePaLambda()` to transform the "selected" loop nest. Whether or not a loop can be transformed is determined by `makePaLambda()` or `pt_pedLambda()` below it. As some feedback to the user, `makePaLambdaAll()` counts the number of disjoint loop nests and the number of transformed loop nests, and reports the results as a percentage of "successful cases" of transformations.

A number of improvements were added in migrating `pala.C` to `pedLambda.C`. One was the replacement of the `./distribution` file, already discussed in Section 5.2.2. The other was to make the set of transformation choices extendible, by adding a C switch, controlled by `lambda_kind_val`. Current selection cases include data access normalization and reading a matrix from a file.

Major simplifications to the `pedLambda.C` code came from improved facilities in the second Lambda Toolkit release. For example, new functions to allocate and deallocate Lambda data structures rendered unnecessary many lines in the former `pala_glue`.

We were able to produce simpler expressions in the transformed loop nest because AST editing facilities were available in PEDX that were not available in PALA. The utility functions underlying `pedX` include a small collection that work together to accomplish some integer, linear expression simplification. This was just the ticket for improving our Lambda-transformed loop bounds and subscripts.

Finally, moving the Lambda-based transformation from a one-loop batch processor to an interactive editor forced us to add more robust error handling and more graceful recovery from cases that couldn't be handled. Again, the new Lambda Toolkit allocation/deallocation functions helped rationalize memory management for our working storage.

Appendix B provides more details on some functions that implement the PEDLAMBDA augmented X based dialogs, and the data and control flow between the new user interface and transformation actions.

6 *The Integration of* PEDLAMBDA *and FORGE*

In the previous section we have described the evolution and use of the new transformation tool, PEDLAMBDA. Having used PEDLAMBDA to optimize a program for parallel computation, the next step is to feed the optimized code to a parallelizer.

We use FORGE to generate a message passing program for the SP1. Because the loop nests presented to FORGE are already normalized for localized data access on a distributed memory machine, FORGE is able to generate very efficient message parring code.

We mainly use a combination of FORGE 90 DMP and xHPF, touching up the xHPF parallelizations with interactive sessions in DMP. The latest DMP shares analysis information with xHPF — one can move from one to the other (manually) during an analysis and transformation process for a program. The advantage of xHPF is that partitioning and distribution decisions can be recorded in the form of directives inserted in the original Fortran 77 program. These will be ignored by PEDLAMBDA, but will be heeded by xHPF in producing a parallel distributed memory program. The advantage of using DMP is that one can interactively examine variable uses and control flow in the program, and choose to alter some decomposition decisions or communications. FORGE is conservative, and occasionally inserts communication unneeded to keep data coherent.

The SPMD program that results from running xHPF (or DMP plus pref77[6]) is linked with SP Series MPL or MPLp runtime libraries and run on the SP. See Figure 10 for a flow chart of how one might use the integrated tool system for migrating from unparallelized, unoptimized Fortran 77 code to a parallel program for the SP Series machine. The next section briefly describes one experience using the combined system.

7 *Data Access Normalization: A Case Study*

We used Data Access Normalization to parallelize a real application: an economic policy code, MPROJTARG, that models world trade under various tariff structures [3]. Table 1 lists some of the characteristics of this program and some datasets. We used FORGE to generate a message passing program based on a given array decomposition for the serial program. Before data normalization, the 4-way PVM version of the DMP–parallelized program ran for three hours and the MPL version died before completion. Running a partial run in MPL and extrapolating from that, we found that the complete DMP-generated program would probably have run for at least two hours. These

[6]The user's session with DMP produces a database of annotations about how to parallelize the source program. The actual generation of the parallelized Fortran source, after a DMP session, is done by invoking the FORGE tool named "pref77". Its code-generation function is built directly into the xHPF tool.

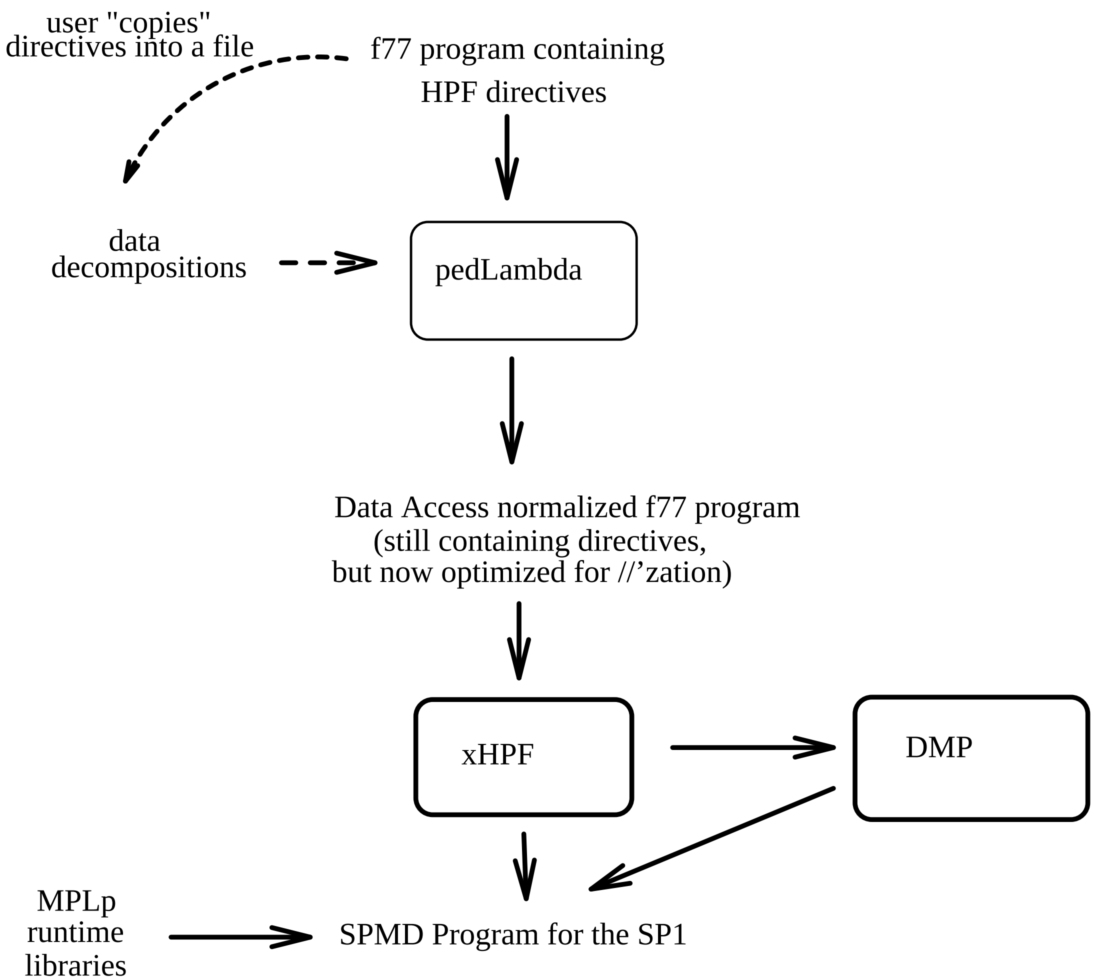

Figure 10: How pedLambda output is processed by xHPF to produce an SPMD program for the SP1. Actually, this is a rather simplified diagram. Both xHPF and DMP produce an SPMD program containing calls to APR's runtime routines. One can compile and link this program with the MPLp versions of IBM's and APR's runtime libraries or alternatively, one can choose the MPL versions, or even the PVM versions. The best results are obtained with the MPLp versions.

PROGRAM & DATA CHARACTERISTICS

	nsupm	ndemm	itcnt	itc	perc	serial time on the SP1
Dataset 1	100	100	4611	194	1.94	1–3 minutes
Dataset 2	200	200	2951	566	1.415	7 minutes
Dataset 3	300	300	2796	153	.1700	14–15 minutes
Dataset 4	400	400	4140	33	.0206	52 minutes
Dataset 5	500	500	2825	7	.0028	52–53 minutes

```
nsupm = number of supply markets
ndemm = number of demand markets
itcnt = number of iterations to convergence, plus 1
itc = the number of non-zero elements at the end of the run
perc = itc / (nsupm*ndemm) = % of elements that are non-zero
```

Table 1: Characteristics of the five datasets available for the MPROJTARG code. Note that the number of iterations to convergence is data dependent, as well as problem size dependent. The program has converged when none of the `nsupm`×`ndemm` elements in the system have changed since the previous iteration. The exact time of the serial run varies depending on system load and compile time options.

runs used a small dataset (the 100x100 Dataset 1, see Table 1); performance for the other datasets would have been even worse. Looking at a program trace using IBM's VT trace visualization tool, we saw an abnormally large number of communications at every step of the program. The tracefile was about 20 MBytes long and took about four hours to replay at top speed!

To fairly evaluate the true effect of data access normalization, we note that many of the communications in the initial FORGE-generated SPMD program were unnecessary.[7] By using FORGE 90 DMP to delete some data exchanges, runtimes crept into the feasible range, but were still quite high. Table 2 shows that it could still take the better part of an hour to complete a run.

Then we used PEDLAMBDA to apply data access normalization. The resulting program was again parallelized by FORGE and run on the SP1. The effect of data access normalization was stunning, especially for the 500x500 dataset. Normalization improves its 4-way runtime by around

[7]DMP and xHPF, with no other advice from the user, are appropriately conservative in deducing the need for communication to accommodate inter-loop-nest dependence assumptions. The use of "indirect addressing" in MPROJTARG obscures the true simplicity of the dependence situation.

UNNORMALIZED PROGRAM

	serial	1-way	2-way	3-way	4-way
Dataset 1	2:18	10:16	10:22	10:12	11:13
Dataset 2	8:09	18:39	18:43	18:40	18:45
Dataset 3	19:02	34:47	33:42	33:47	33:57
Dataset 4	52:27	80:19	80:29	80:13	80:19
Dataset 5	52:13	77:08	76:58	77:04	77:04

Table 2: SP1 runtimes of the parallel version after unnecessary communications were deleted, but before data access normalization was performed. All communication was done over the SP1 switch, and the parallel program was compiled with `-ip` and `-O2`. Elapsed times are given in min:sec. "Serial" denotes the f77 equivalent of the parallel program, without any parallel overhead. "1-way" denotes the parallel program being run on a single processor, so there is no communication overhead, but some parallel overhead may be seen.

70%. Table 3 contains the runtimes of the data normalized program.

What happened is that before normalization, inter-processor data exchange took place inside loop nests. Since MPL latency on the SP1 is rather high, each message inflated run time. Thousands of messages were sent. Even after removing unnecessary communications, there are still $O(N^2 + \log N)$ messages being exchanged by the program for data size N, for each of `itcnt` iterations. After normalization, this was reduced to a single size N transfer and a single $\log N$ data-reduction per iteration.

Comparing column 1 of Table 2 with column 1 of Table 3, we see that data access normalization improved the runtime even of the serial program. That's because in this particular case, data access normalization corresponded exactly to the optimization one would do to improve cache coherence. In fact, the KAP preprocessor for IBM's XLF compiler interchanged many of the same loops that data access normalization did, but for different reasons. (See [3] for complete details.)

The other thing to note from Table 3 is that data access normalization by itself did not provide parallel speedup. The reason for this is that for FORGE generated programs, all file I/O is handled by a single instance of the SPMD program; input data is then broadcast to each of the other instances. This takes more and more time as the number of processors increases. By parallelizing file I/O, we got decent speedup on up to 8 processors for the data access normalized code (see Table 4). To get even better performance, one would next have to find a way to reduce the total

NORMALIZED PROGRAM

	serial	1-way	4-way
Dataset 1	1:04	1:19	4:33
Dataset 2	2:41	3:07	6:52
Dataset 3	5:38	6:20	19:38
Dataset 4	17:54	15:56	30:24
Dataset 5	15:25	16:57	20:34

Table 3: SP1 run times after data access normalization. Again "serial" is the reorganized serial program, and "n-way" is the message-passing program run on n processors. Data access normalization achieves up to 80% improvement over the unnormalized code.

NORMALIZED PROGRAM WITH PARALLEL I/O

	1-way	2-way	3-way	4-way	5-way	6-way	7-way	8-way
Elapsed Time	15:40	8:45	6:51	6:51	7:07	4:51	4:32	4:23
CPU Time	779	431	314	252	220	197	180	167

Table 4: Run times of data access normalized program with parallel file I/O, compiled with MPLp and run on the SP1, for the 500x500 dataset (Dataset 5). Elapsed Time is in min:sec and includes a manual control-C operation to stop the prototype program launcher available for MPLp. The CPU Time is in seconds and is node 0's CPU time only. The program ran in a serialized batch queue.

number of iterations, an algorithmic change beyond the scope of this paper.

8 Conclusions and Future Work

We believe that we have designed and developed a usable data access normalization tool by integrating the work of other developers. This tool uses the strengths of each, and works together more or less seamlessly to provide a powerful parallelization tool. The major limitation is the need to input decomposition data by file, rather than by directives.

We have found that PEDLAMBDA and FORGE 90 DMP/xHPF can be coupled to other tools in the parallel programming environment. Recently we have been gaining insight into the performance of parallelized codes by examining their communication patterns in detail. Despite our not coding any calls to the communication functions "by hand" when using the combined PEDLAMBDA-FORGE, we have been successful at collecting communication trace records on the SP1 from codes loaded with either the MPL or MPLp communication libraries. These trace records can be examined with the IBM VT trace visualization tool [9] or the PABLO trace analysis system[19, 20]. The results have led to tuning the behavior of the parallelization process with significant improvements in performance (see Section 7).

There are several ways in which this work could be extended. Certainly more transformations could be added. A new version of the Lambda Toolkit is expected which will accept imperfect loop nests. The PEDLAMBDA tool ought to be extended to use this new technology. It would be desirable to handle decomposition information directly as HPF directives. Finally, this tool should be deployed to users: to do this it must be migrated to the CTC AFS-based RS/6000 cluster, and user documentation must be provided.

9 Acknowledgements

This research was conducted using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation, and New York State. Additional funding comes from the Advanced Research Projects Agency, the National Institutes of Health, IBM Corporation and other members of the center's Corporate Research Institute.

We continue to acknowledge the help of Alan Carle and John Mellor-Crummey of the Center for Research on Parallel Computation, Rice University, for pointing out to us the right fulcrums and levers inside the ParaScope software mechanism, and the work of Professor Wei Li, of the University of Rochester, for the Lambda Toolkit software. We wish to thank Claus-Peter Wiedemann, Siemens, Germany, for another valuable "Rosetta Stone" in aid of our understanding of how ParaScope can be extended. We wish also to thank Keshav Pingali of Cornell for originally suggesting this project.

References

[1] G. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Co., Inc, Redwood City, CA, 1994. Second Edition.

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers: the Foundations.* Kluwer Academic Publishers, 1993.

[3] D. Bergmark and M. Pottle. Optimization and parallelization of a commodity trade model for the SP1. Technical Report CTC94TR181, Cornell Theory Center, 1994.

[4] D. Bergmark and D. Presberg. Initial experiments in the integration of ParaScope and Lambda. Technical Report CTC93TR136, Cornell Theory Center, June 1993.

[5] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–89, Winter 1988.

[6] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–459, 1989.

[7] J Dongarra, G.A. Geist, Robert Manchek, and V.S. Sundaram. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, pages 166–175, 1993.

[8] S. Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8), August 1992.

[9] IBM. *AIX Parallel Environment Parallel Programming Reference*, September 1993.

[10] IBM. *IBM Scalable POWERparallel Systems Reference Guide*, 1993.

[11] Intel Corporation. *iPSC/860 Fortran System Calls Reference Manual*, March 1992.

[12] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[13] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 136–152. Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993. Also available as technical report CRPC-TR93-298-S, Rice University.

[14] W. Li. Compiler optimizations for cache locality and coherence. Technical Report CS Technical Report 504, University of Rochester, April 1994.

[15] W. Li. The Lambda loop transformation toolkit (user's reference manual). Technical Report Computer Science Technical Report 511, University of Rochester, May 1994.

[16] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. *ASPLOS '92*, 1992.

[17] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. Technical Report CTC92TR99, Cornell Theory Center, July 1992.

[18] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proc. 5th Annual Workshop on Languages and Compilers for Parallelism*, August 1992. Also published as Cornell Theory Center Tech Report CTC92TR98, July 1992.

[19] A. D. Malony and D. A. Reed. Visualizing parallel computer system performance. Technical Report UIUCDCS-R-88-1465, csrd, September 1988.

[20] A. D. Malony, D. A. Reed, J. W. Arendt, R. A. Aydt, D. Grabas, and B. K. Totty. An integrated performance data collection, analysis, and visualization system. Technical Report UIUCDCS-R-89-1504, Center for Supercomputing Research and Development, U. of Ill., March 1989.

[21] ParaSoft Corporation, Pasadena, CA. *Express 3.0 Introductory Guide*, 1990.

[22] Applied Parallel Research. *FORGE 90 Baseline System User's Guide*. 550 Main Street, Suite I, Placerville, CA 95667, June 1993.

[23] Applied Parallel Research. *FORGE High Performance Fortran xhpf User's Guide*. 550 Main Street, Suite I, Placerville, CA 95667, December 1993.

[24] Applied Parallel Research. *FORGE 90 Distributed Memory Parallelizer User's Guide*. 550 Main Street, Suite I, Placerville, CA 95667, April 1994.

[25] R. Stefani. IBM falls in parallel with 9076 SP1. *High-Performance Computing Review*, April 1993.

[26] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, pages 315–339, December 1990.

[27] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

APPENDIX A: Example of loop transformation

```
                  Initial Iteration       Transformation
Initial loop            Space             to be Performed
------------      -----------------       ---------------
do i = 1,3              o o o             (loop skewing)
   do j = 2,3           o o o                +-   -+
                                             | 1 2 |
     a(i,j) =                                | 0 1 |
                                             +-   -+


 New, desired      Corresponding
Iteration Space      Loop Nest
----------------   -------------
         o         do u = 1,3           <== Lambda toolkit
         o            do v = 2u+2,2u+3   computes new loop
       o                                 bounds automatically
       o                a(?,?) =
     o
     o
```

```
The question is, what should the new subscript expression for
the original array reference, "a(i,j)" be??  Simply invert
the transformation matrix (using the Lambda toolkit):

     +-    -+ -1          +-    -+
     | 1  2 |             | 1 -2 |
     | 0  1 |        =    | 0  1 |
     +-    -+             +-    -+

Then apply it to the subscript expression:


      +-    -+            i.e. "i" <- u
      | 1 -2 |                 "j" <- v-2u
<u v> | 0  1 | = <u v-2u>      so,
      +-    -+                 a(i,j) becomes a(u, v-2u)
```

APPENDIX B: Code Changes to PEDX *for* PEDLAMBDA

The PEDX "control program" library, `ped_cp`, is built from two subtrees of source files, `dp` and `pt`. (Refer to Figure 9 in Section 5.3.) On the `dp` side, the major modification was to the "interface-describing" (and enabling/disabling) code in `trans_menu.c`. A ParaScope interactive tool menu is built hierarchically with utility function invocations that return "handles" to data structures that "manage" the menu-selection system. One can embed various kinds of fields or choice-buttons, one-within-another, by composing (applicative-function style) one interface-describing function call as an argument of another. Equivalently, one can provide variables to hold the constructed handles of lower-level structures (inner menu or button items) and then pass those handles as actual arguments to the calls that construct a higher-level structure in the hierarchy. The ParaScope infrastructure manages the actual display of menus, buttons, and other visible fields, elsewhere in its code — we never had to learn the detailed mechanisms. (We note that, although there is some C++ used in ParaScope, the part of the user interface code that we modified was coded in ANSI C, albeit with a strong "Object-Oriented, message-passing, paradigm" flavor.) The following is the entirety of the code that constructs the `Lambda-Based Preferences...` dialog menu depicted in Figure 6:

```
void
lambda_dialog_run(lambdah)
     AllDia *lambdah;
{
  Dialog *di;

  di = dialog_create(
                     "Lambda-Based Preferences ...",
                     lambda_pref_handler,
                     (PFV) 0,   /* no help; too obvious */
                     (Generic) lambdah,
                     dialog_desc_group(
                                DIALOG_VERT_CENTER,
                                2,
                                item_title(
                                       UNUSED,
                                    "Select kind of Lambda-based Transform:\n",
                                       DEF_FONT_ID),
                                item_radio_buttons(
                                       LAMBDA_KIND,
                                       DIALOG_NO_TITLE,
                                       DEF_FONT_ID,
                                       &lambda_kind_val,
                                       1, 3, /* cols, rows */
                                       "Access Normalization",
                                                         L_ACCESS_NORM,
                                       "Read Transform Matrix from File",
```

```
                                                                  L_ANOTHER_1,
                                                "Another Kind 2", L_ANOTHER_2
                                                            )
                                                )
                              );

  dialog_item_ability(di, LAMBDA_KIND, DIALOG_ENABLE);
  dialog_modal_run(di);
  dialog_destroy(di);

} /* end of lambda_dialog_run */
```

The dialog menu consists of a "group" of 2 lower-level items, centered on a vertical line, the first being a "title" in the "default" font, the second being an item containing "radio buttons" (one-of-N exclusive choices). The radio button item contains 3 choices, arranged vertically (one column of 3 rows), each containing text, such as `Access Normalization`. When a choice is selected, its corresponding value from an enumeration type, such as `L_ACCESS_NORM`, is stored in the `lambda_kind_val` global variable. Another function, `lambda_pref_handler`, passed as an argument to `dialog_create`, deals with the "confirm/cancel" protocol available in any ParaScope interactive tool dialog. Static initialization of the `lambda_kind_val` variable selects the default choice value.

The major change to `trans_menu.c` was the addition of the `lambda-based` choice to the `Transformations` menu (Figure 5) to conduct the dialog, enable new menu items for loops meeting certain criteria, and respond to transformation selections. The actual code in `trans_menu.c` that "signals" from the `dp` side that the user wants one selected loop nest to be transformed is an invocation of the function `makePaLambda(ped)`, defined in `trans.c` on the `pt` side:

```
AST_INDEX                       /* return value will be used in ...All */
makePaLambda(ped)
     PedInfo ped;
{
  AST_INDEX       scope;
  AST_INDEX       selected_loop;

  scope = find_scope(PED_SELECTED_LOOP(ped));
  selected_loop = PED_SELECTED_LOOP(ped);
  if( pt_good_nest( selected_loop ) <= 1 ) /* 0 ==> not tight nest */
                                           /* 1 ==> only 1 DO loop */
                                           /* >1 == depth of tight nest */
    {
      message(
"Can only transform perfect DO loop nests\n   that have more than one DO.\n"
             );
      return AST_NIL;           /* indicates not transformed to make...All */
    }
```

```
  ped->TreeWillChange(PED_ED_HANDLE(ped), scope);

  /* Attempt to transform one nest */
  if( !pt_pedLambda(lambda_kind_val, ped, PED_SELECTED_LOOP(ped)) )
    return AST_NIL;             /* indicates not transformed to make...All */

  ped->TreeChanged(PED_ED_HANDLE(ped), scope);
  ped->UpdateNodeInfo(PED_ED_HANDLE(ped), PED_SELECTED_LOOP(ped),
                      UPDATE_SUBS | UPDATE_DTINFO);
  forcePedUpdate(ped, PED_SELECTED_LOOP(ped), PED_SELECTION(ped));

  return (PED_SELECTED_LOOP(ped));
}
```

Note that `makePaLambda()` tests a returned Boolean result from `pt_pedLambda()`. This provides a facility to gracefully back out of attempts to transform a loop nest that doesn't satisfy criteria needed for Lambda Toolkit transformations, such as having loop bounds more complicated than linear expressions of outer DO variables. The `lambda_kind_val` variable, defaulted or set by the `Preferences...` dialog, is passed as an argument to `pt_pedLambda()`, as are two data structures that provide access to the single loop nest in the AST that the user (or other `dp` code) has selected. The `lambda_kind_val` value is used in the C `switch` that selects the particular Lambda-based transformation (see Sections 5.2.2 and 5.3).