Integration of ParaScope and Lambda

Donna Bergmark and David Presberg
Cornell Theory Center

May 19, 1993

Abstract

This document describes the work we did for the CS612 project. Our project was to incorporate Wei Li's Lambda toolkit into the ParaScope parallel programming environment. The goal was to improve the functionality of ParaScope, to determine the usefulness of Lambda in environments other than that of its original development, and to evaluate the quality of code generation before and after incorporation of Λ-based analysis. We learned that ParaScope could be extended, but only by very brave people; we learned that the Lambda Toolkit could be used by other tools to good effect. We also compare two different proposed interfaces for the Lambda Toolkit.

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

Contents

1 Introduction

The general goal of this project was to integrate two different compiler projects meant to enhance general scientific programming, especially in parallel and distributed computing environments. We have the source for ParaScope, a Scientific Programming environment under development at Rice University as well as the Lambda toolkit, under development at Cornell. Since the toolkit needs an environment, and since ParaScope lacks some of the function of lambda, the combination seems to benefit all. At the same time, the project fulfills the valuable function of evaluating the extensibility and usefulness of ParaScope, while also providing an "alpha test" of the Lambda Toolkit. Both are described in more detail below.

1.1 ParaScope

Parascope is the outgrowth of CRPC/Rice University research into advanced programming environments for scientific programming in Fortran[4]. The current release of ParaScope sources organizes it, less as an all-encompassing programming environment, than as a collection of mutually compatible program-development and transformation tools. Some of these, notably PED, the ParaScope Editor[7, 8], are built with an X-based graphical user interface that promotes interactive exploration of dependence analysis, reporting attributes of variables (such as definitions and uses controlled by *view filtering*), and editing and transformation of the source code to enhance parallelism. In addition, the current release of ParaScope contains examples of non-interactive Unix filters to do Fortran source-to-source dialect translation (e.g., FORTXFORM) and other transformations that are easily derived from the source text (possibly with inserted *directives*) or driven by deep analysis of the entire program context of the source module being processed.

The current release of ParaScope ("ParaScope 1.0 distribution tape" of early April 1993) is understood by its developers to contain prototypes of the tools that ultimately could be built. For instance, although the Fortran lexical analysis and parsing components accept a large dialect of Fortran 77 (include at least 3 different manufacturer's extensions that denote shared-memory parallel execution semantics), the developers have made no attempt to validate ParaScope against standard Fortran 77, and there are known omissions. (In addition, various popular extensions to Fortran that appear regularly in industry-popular benchmarking suites may not be handled correctly by ParaScope.) In addition, the Rice developers had fairly little interaction, involving the source code for ParaScope, with any other researchers attempting to extend or modify their efforts. A result of both the prototype and isolated development attributes is that there is almost no documentation of the internal organization and implementation details for large parts of the system.

The ParaScope prototype transformation tools are not, per-se, compilers: they are really editors. A result of this design principal is that the internal representation of a program, although it has aspects of conventional compiler data structures (e.g., an Abstract Syntax Tree), is closer to the source text of the program, later in the analysis than one ordinarily finds in a machine-code generating compiler.

1.2 Lambda Toolkit

The Lambda Toolkit developed by Wei Li uses a non-singular matrix to model transformation of loop nests.[9, 10, 11, 12] This is an important extension of Bannerjee's work [3] who proposed a unimodular matrix for this purpose. There are many optimizations in parallel and scientific computing that are based on transformation of loop nests (see Appendix B). Using integer matrices has the big advantage of avoiding the problem of deciding in what order the transformations should be made.

When we started this project, the toolkit was available in source form from the Computer Science Department's network of SUN workstations; it consisted of a library (`lambda.a`) and an include file (`lambda.h`). There were also some example programs. The library's routines performed the work described in [11] and in [12]. We copied this over to our group's workstation and proceeded to write routines using the toolkit. Then we called these routines from ParaScope.

Lambda's completion procedures are not particular about what transformation is specified, so long as it is invertible. We chose to concentrate on the Data Access Normalization transformation, since that would be of most interest to Theory Center users of the RS/6000 parallel cluster and to the recently announced SP1 [15]. This transformation reorganizes loops so as to group together references to distributed array variables together such that the data can be communicated in larger chunks, less frequently. This is described in the ASPLOS paper by Li and Pingali [11].

1.3 FORGE

Once ParaScope has called Lambda to reorganize the loop nest, it outputs the new Fortran code which can then be fed into FORGE 90 [13] and manipulated there. The output of FORGE 90 is a program with message passing to handle array decomposition and loop distribution. It can be run on top of PVM [5, 16] and SP1 communication libraries.

The essential component of FORGE 90 that performs the message-passing insertions is the Distributed Memory Parallelizer (DMP). This is an interactive tool that parses and analyzes a source module in the context of an entire Fortran 77 program. The user indicates to DMP particular `DO` loop nests for (inter-procedural) dependence analysis and potential *loop distribution*. After viewing the dependence analysis results and other indications of required communication for potential loop distributions, the user accepts the suggested loop distribution or rejects it. FORGE 90 DMP uses an *owner computes* rule to perform loop distribution and *data decomposition*. Along with loop distribution, the FORGE 90 DMP user must indicate preferred data decompositions, either block or cyclic, array by array, at each loop or subroutine of interest. (The DMP runtime support does bookkeeping of the current, dynamic data decomposition situation at each loop or subroutine that uses distributed data. Since DMP's static analysis has the entire call-tree of the program, some optimization is done automatically, and also the user can count on decompositions high in the call-tree that cover numerous distributed loops.) Loops that are not distributed are executed serially by all processors, and data that is not decomposed is replicated across all processors.

```
subroutine fig1a( A, B, N1, jb, N2 )

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

integer i, j, k

DO i = 0, N1-1
   DO j = i, i+jb-1
      DO k = 0, N2-1

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

      ENDDO
   ENDDO
ENDDO

end
```

(a)

```
subroutine fig1c(A,B,N1,jb,N2)

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

integer u, v, w

DO u = 0j, jb-1
   DO v = u, u+N1+N2-2
      DO w = 0, N1-1

      B(w,u) = B(w,u) + A(w,v)

      ENDDO
   ENDDO
ENDDO

end
```

(b)

```
     1:         subroutine fig1a( 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:
     8:
   ---> Distribute the loop on B<:, 0~1>
   ---> Preloop  communication of  B<0:n1-1, 0~1>
   ---> Preloop  communication of  A<0:n1-1, :>
   ---> partition A (SHRUNKCYCLIC,*) move
    13:         DO u = 0, jb-1
    14:            DO v = u, u+N1+N2-2
    15:               DO w = 0, N1-1
    16:
    17:                  B(w, u) = B(w, u) + A(w, v)
    18:
    19:               ENDDO
    20:            ENDDO
    21:         ENDDO
    22:
    23:         end
```

(c)

Figure 1: Case example from Wei Li, (a) before and (b) after loop normalization. The idea is that block data transfer of matrix A can be used. The problem is, will FORGE recognize this when asked to parallelize this loop nest? In (c) array A was decomposed in index 1 (i.e. by rows) and the outermost loop was parallelized. To our initial dismay, FORGE decided to replicate A, rather than generate block data communications.

2 Lessons Learned

2.1 Data Communications

Our first problem was whether we could use FORGE to insert the message passing code for us, rather than try to make ParaScope generate message passing code. As a proof of concept, we took Li's data access normalization example as described in [11] and fed it into FORGE and asked for the loops to be parallelized. The results are shown in Figures 1(a) and 1(b).

To our initial dismay, FORGE did not generate the same message passing code as shown in [11]'s Figure 1(d). FORGE successfully determined that the array A should not be communicated piecemeal inside the innermost loop, but rather than a block transfer FORGE to our surprise decided to transfer array A in its entirety (see Figure 1c) before entering the nest. But analysis of the code indicates that in the absence of memory restrictions, FORGE made the right decision. Rather than transferring one column of A `jb` times for each distinct value of `v`, it is better to distribute the entire array once and be done with it, since A is unchanged in this loop nest.

But now we were worried that FORGE would handle only outermost loop parallelization, whereas to test the Lambda Toolkit, we would have to be able to generate intermediate block transfers as well. But we were able to demonstrate that this could be done (see Figure 2). When we asked for array A to be decomposed on the *second* index and for the *middle* loop to be distributed, we did indeed get a potential block communication of A. (Since A is not modified by the loop in this trivial example, FORGE would not actually do the communication inside the loop, having done the decomposition before the loop, but that is irrelevant to this experiment).

At any rate, we had determined that FORGE would do what we want and we turned to the modification of ParaScope and the writing of routines to use the toolkit for data access normalization.

2.2 The Extensibility of ParaScope

ParaScope development is organized over a very large collection of source files, contained in at least 40 to 60 subtrees (we have never counted them), in the mixture of C++ and C described earlier, and managed by a set of scripts and recursive gnumake-files that understand the trees and inclusion relations. Provision is made for a developer's sandbox version of a library or executable. Insertion of new libraries or modules requires careful registration of its position in the structure in a number of controlling texts. (There has been no documentation to date of the exact process.) The current filespace occupied by all ParaScope sources and executables (in `-g` form) is approximately 400 Megabytes of disk space.

ParaScope sources were first brought to Cornell during a visit in January 1993 by one of its developers. Prior to that, we had been briefed at Rice University on the general structure and modularization of ParaScope internals, but there still was no true customization documentation available from Rice. We found that two code reviews were not sufficient to make headway on extending ParaScope code. In early April 1993, Rice released and installed at Cornell the "Version 1.0" distribution.

We also discovered that the programming tools needed to support a system as large as ParaScope are fragile when moved from one development environment to another. Due to circumstances beyond our control, we were forced to move among several different environments.

```
Viewing Parallelization of FIG1C/DO 200020 V

     1:         subroutine fig1c( 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:
     8:
   ---> partition A (*,SHRUNKCYCLIC) move
    10:         DO u = 0, jb-1
   ---> Distribute the loop on A<:, u~1>
   ---> Preloop  communication of  A<0:n1-1, u~1>
    13:            DO v = u, u+N1+N2-2
    14:               DO w = 0, N1-1
    15:
    16:                  B(w, u) = B(w, u) + A(w, v)
    17:
    18:               ENDDO
    19:            ENDDO
    20:         ENDDO
    21:
    22:         end
```

Figure 2: Demonstration that FORGE will be 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. FORGE will bring in all the rows of A and the appropriate columns for v, i.e. u through $u + N1 + N2 - 2$

The initial focus for development work was based on an RS/6000 AIX system. The versions of pieces of the g++ support environment present on the RS/6000 were at variance with those in use at Rice and not comparable to those in use on Sun platforms. Early in February 1993 development was moved to a clustered RS/6000 with a different view of standard Unix program development tools than had been used during the initial installation from sources of January 1993. For the next two months, we encountered a sequence of incompatibilities, system failures, and software mis-matches. Since our effort was moved to a Theory Center Sun4 platform, progress has been rapid.

Currently, the developers of ParaScope understand the need for (and absence of) documentation of its internal structures, libraries, interfaces, and development procedures[14]. Our technical contact at Rice provided working, pedagogical, code examples that exemplified the functions used in the ParaScope framework for traversing an AST, and accessing fields and attributes of AST nodes, among many others. We noted that the top-level action skeleton of a ParaScope tool is obscure: the `main()` function only serves to launch an environment that processes command-line arguments and opens files, etc., via a demand-driven implication network. We have extended one of the pedagogical code samples as the framework for our PALA executable.

Considerable that had to be done to extract the information that was needed for the Lambda Toolkit. For instance, one fundamental data requirement for the Lambda transformations is the coefficients of a linear-form array subscript expression. This data is also commonly used in dependence analysis: there is a dependence analysis package in ParaScope. Unfortunately, the particular linear expression analyzer already present in ParaScope was buried so deep in a heirarchy of other modules, and was so specialized to the dependence analysis effort, that it could not easily be employed for the code of this project. We thus wrote the more or less obvious mutually recursive AST walking routines to find and extract the coefficient values of `DO` index variables in linear array subscript expressions.

Finally, finding the right mixture of C and C++ linguistic expression to satisfy the particular compilers and support structure (they are installed with the ParaScope distribution) has been an experimental process. The coding style in `pala.C` has been that of C with standard C structures, rather than C++ classes and member functions. Various C++ or ANSI C cliches have been employed where they could be seen to work in pre-existing code samples. The code has been written with a view to debugging, rather than for raw performance of the tool.

That we could create a PALA demonstrates that ParaScope can be extended.

2.3 The alpha Test of the Lambda Toolkit

It was a very good thing that the lambda directory contained a sample program that called the lambda library. It took about a week to figure out what this program was reading in, what it was feeding to the toolkit, and what it was getting back out. The existing user's guide was not helpful (it was an early draft), but face to face chats with Wei Li were.

Having figured out how to run a sample program, we wrote a document describing the file formats used by it. The sample program hands a transformation, a loop nest, and a list of dependences to `la_bounds` and gets back new bounds, as well as a new loop transformation. This should save somebody else the week of two we spent feeling our way around.

Once the cultural acclimation to the Lambda Toolkit was made, we found that using the lambda

library was very straightforward. Only a handful of bugs were found, and reported. The current library does output too much stuff to `stdout`, even when the global print flag is off, but that was more helpful than not in our early stages of debugging.

One thing we learned during figuring out the sample program is that the lambda toolkit expects the user to supply his/her own transformation matrix. We therefore wrote a routine called TGEN to produce a transformation based on the data access normalization techniques describe in [11]. This routine is given a set of dependence vectors and subscript expressions along with a set of flags indicating whether each subscript is in a distribution dimension or not. The subscript expressionss are rearranged and sorted and duplicates are removed. Then the completion process is performed by calling `la_complete`. This procedure extracts a basis matrix from the subscript expressions, and pads them out to a legal, non-dependence-violating, invertible transformation matris.

The next thing we discovered was that the lambda toolkit did not handle loops with steps other than 1. We furthermore learned that ParaScope also did not contain this transformation. While everyone agreed that the routine could easily be written, nobody had yet done it. So, since non-unit steps are very common in scientific Fortran programs (for unrolled loops, for red-black algorithms, etc.), we decided to write our own loop normalization routine (NORM). This took about a week and some heavy matrix algebra. It was actually more complicated to write than one would first believe.

Recommendations for the lambda toolkit:

- Allow the input of complicated loop bounds
- Handle step sizes other than one
- Provide some sample transformation-generators
- Include several examples with the software distribution

2.4 Interfaces

Toward the end of the project, the lambda interface was redesigned. We chose to use this new interface in ParaScope and write a separate module (PALA-GLUE) to translate from the new data structures into the old data structures before calling the lambda library routines. This routine turned out to be fairly sizable. We by and large used the existing lambda routines to print out the data structures. We anticipate more complete print routines once the new interface is complete.

Comparing the new interface with the old interface we found that the new interface is much easier to pass between routines (many fewer arguments) but was complicated to manipulate. However, we had to write all our own allocation routines. The new library should provide all the necessary allocation routines, output routines, and access functions, so this complexity will be eventually hidden from the programmer. In general, we found that the interface specification resulting from the May 6 review meeting were correct (one missing field was discovered). We also found out why the new interface has seemingly unnecessary pointers at the top level, as will be explained shortly.

Another interface difficulty was mixing C++ and C codes together. ParaScope is written in a stylized and carefully managed mixture of C++ (for a particular version of the g++ compilation environment) and C (for the corresponding version of the gcc compiler), whereas the Lambda Toolkit is written in C. It is understandable that Wei Li had no insight into the particular conventions

and dialect restrictions of C that ParaScope imposes. We therefore were initially worried about the potential mismatch of compiler and linker conventions that could have led to mutually incompatible `.c`, `.C` (g++) and `.h` files, or could have produced hard-to-fix misbehaviour of a seemingly good loaded executable.

Being worried about these language problems, our initial approach was to uncouple ParaScope and the Lambda Toolkit as much as possible. ParaScope was to produce text files containing the data to be fed to the Lambda Toolkit; Lambda was to read those files, perform its numerical transformations, and return files of specifications of how the source code was to be transformed; then ParaScope was to read those Lambda-output files and perform the actual transformation of the source code. This led to early development of text-dumping facilities added to ParaScope and a file-reading driver harness for NORM and TGEN. In fact, these routines output files as well.

After some success with this approach, we performed a small experiment in integrating some C code that called the Lambda Toolkit with the evolving ParaScope-based filter, written in C++, but written with a C-like paradigm for control and data structures. The experiment succeeded, and the project now has object code integration of the Lambda Toolkit and ParaScope.

However, we now have analogues of each routine that can be run in a free-standing way for debugging and further code development. The object level integration also showed us why Wei Li designed his new interface to have some extra pointers at the outermost level of the structures: it turned out that our version of g++ would not pass structures to the C code; we had to use pointers to structures.

3 Integration Design

Designing an integrated system raised several design issues. One was how to pick out which of several transformations to use (by default we use data access normalization, but we could also have skewing transformations as in Bannerjee's paper, or even random transformations).

A second interesting problem was whether to normalize the loop first and then do access normalization, or the reverse. We chose to do the former, then recompute all the subscript expressions according to the new loop but left the dependences alone, and then did access normalization. Therefore we first call NORM and then TGEN.

Thirdly, there was the communication between Lambda and ParaScope. For this project, we decided that the effort involved to extend the PED graphical user interface to embed interaction with new transformations provided through the Lambda Toolkit: (1) would require more time than we had available to complete the project, and (2) would not be pertinant to the central goal of the project. Our approach was to build a source-to-source Unix filter-form program that used the essential language-processing parts of the ParaScope framework, and the Lambda Toolkit. At a later date, we can transfer the lambda calls into the PED framework. As also mentioned above, we decided to code in the new interface on the ParaScope side, and the old interface on the Lambda side, with a program called PALA_GLUE in between.

A fourth issue was how to input the array distribution data. Ideally, this should come from the ParaScope environment from the analysis of a Fortran program. Fortran D (a variant of HPF) would allow the user to specify array decompositions in the source file, but since we were unable to get the Fortran D part of ParaScope until late April, we decided to read array distributions from a file.

The top-level view of the system is shown in Figure 3, and the integration scheme we first wanted is shown in Figure 4.

But there were some problems with this initial view, the main one being the difference between the old and new lambda interfaces. So we interposed `pala-glue` as shown in Figure 5.

4 Results and Deliverables

4.1 A New ParaScope

We have developed a new prototype of the ParaScope prototype, which has an additional loop transformation that allows optimized data communications. This new prototype has been tested on some sample codes.

4.2 Examples of Sample Fortran Program Analysis

4.3 Meta Lambda Tools

We have coded a loop normalizer, a data access normalizer, and a loop transformer, all using the lambda toolkit. Each of these tool stands alone as a program, or as a subroutine that can be called from the ParaScope environment (using PALA_GLUE as an intermediate). Each of these subroutines is accompanied by a simple driver to allow unit testing.

4.4 Documentation

We have begun some documentation of the internals of ParaScope, concentrating on what developers will find most useful. We have written a guide on using Wei Li's Lambda Toolkit example. Each of our meta lambda tools is documented as well.

5 Further Work

This has been a fascinating project, and there remains all sort of things we want to do, in carrying on this work. One is, of course, to add the lambda transformation to ParaScope's Transformations menu (see Appendix B for the current menu). Selecting this button would go off and call the lambda routines, and upon return, present the user with a reorganized loop.

It would be nice to build in the array distribution decisions into the Fortran source program so that these could be presented to TGEN automatically. We would like to use the Fortran D front end of ParaScope to do this, rather than inventing our own comment structures or dialogue tool to get these from the user.

Finally, when the new Lambda interface is finished, we will do the conversions required to eliminate PALA_GLUE, and recode the three "meta-lambda" tools to use the new library.

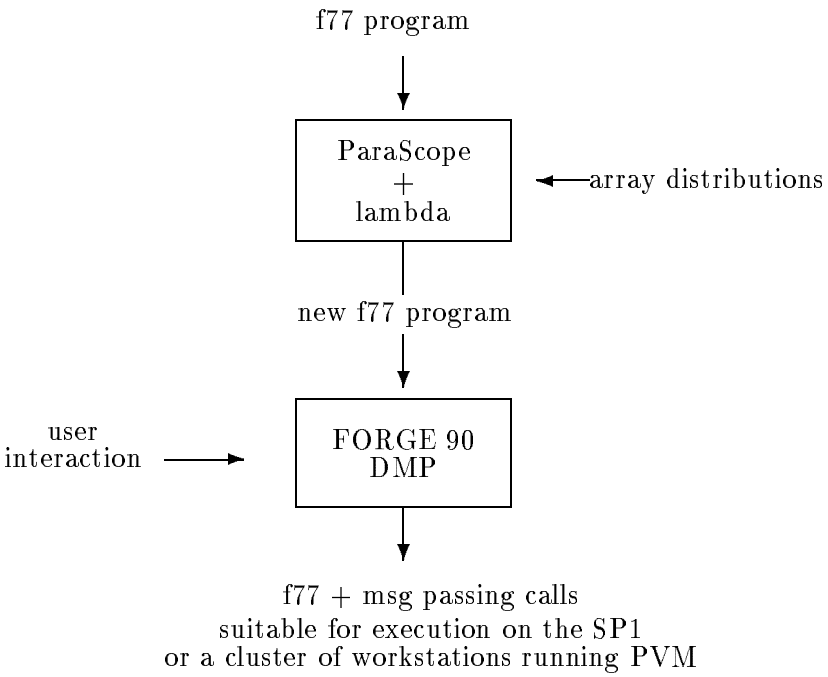

Figure 3: Overall View of a Data Access Normalization using the lambda toolkit, the ParaScope programming environment, and the FORGE Distributed Memory Parallelizer. The user tells Forge which arrays are to be decomposed and which loop to distribute. Eventually ParaScope will allow this to be specified in the input program, when its Fortran D compiler is available.

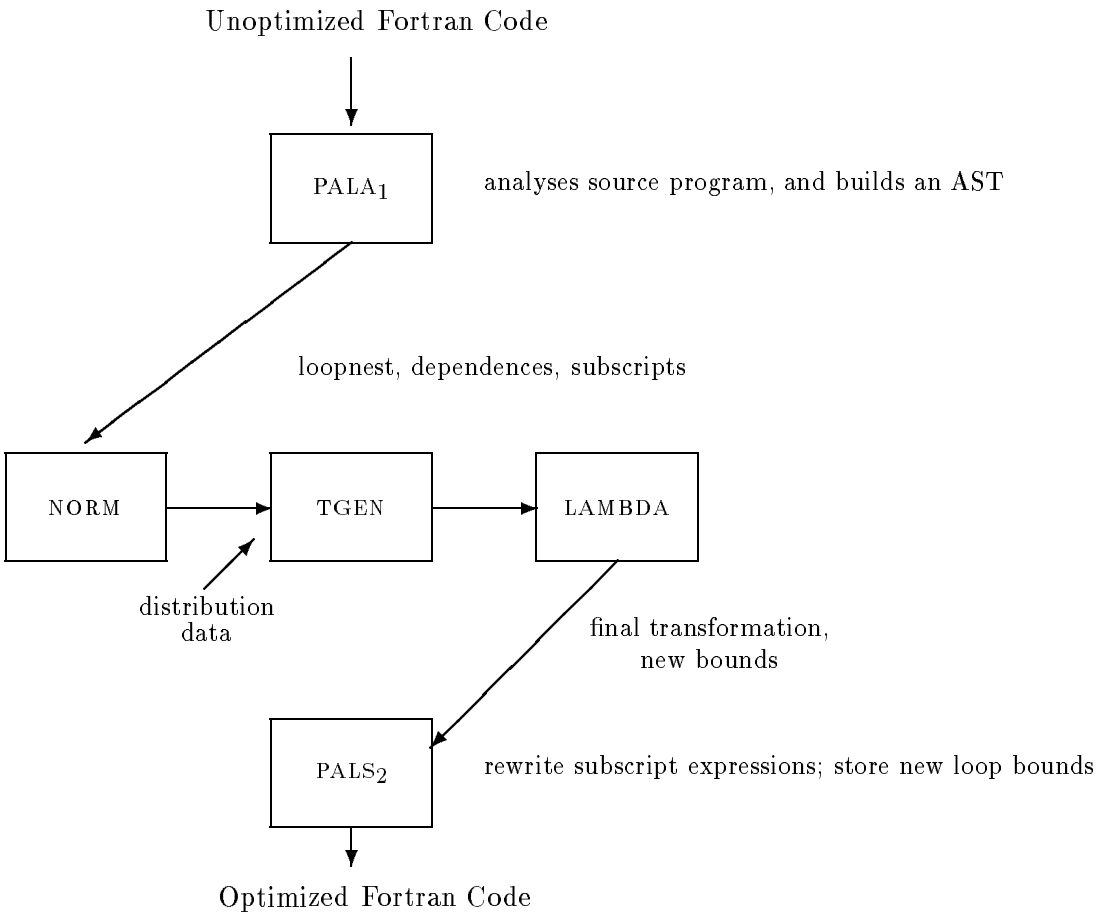

Figure 4: Theoretical Integration of ParaScope and Lambda. PALA is an acronym derived from ParaScope and Lambda. Arrows show flow of control. The two parts of PALA are before and after the call to the Lambda toolkit. The lambda toolkit is called by NORM, TGEN, and LAMBDA. At the moment the distribution data has to be fed into TGEN "on the side" since the Fortran D compiler is not yet implemented in ParaScope.

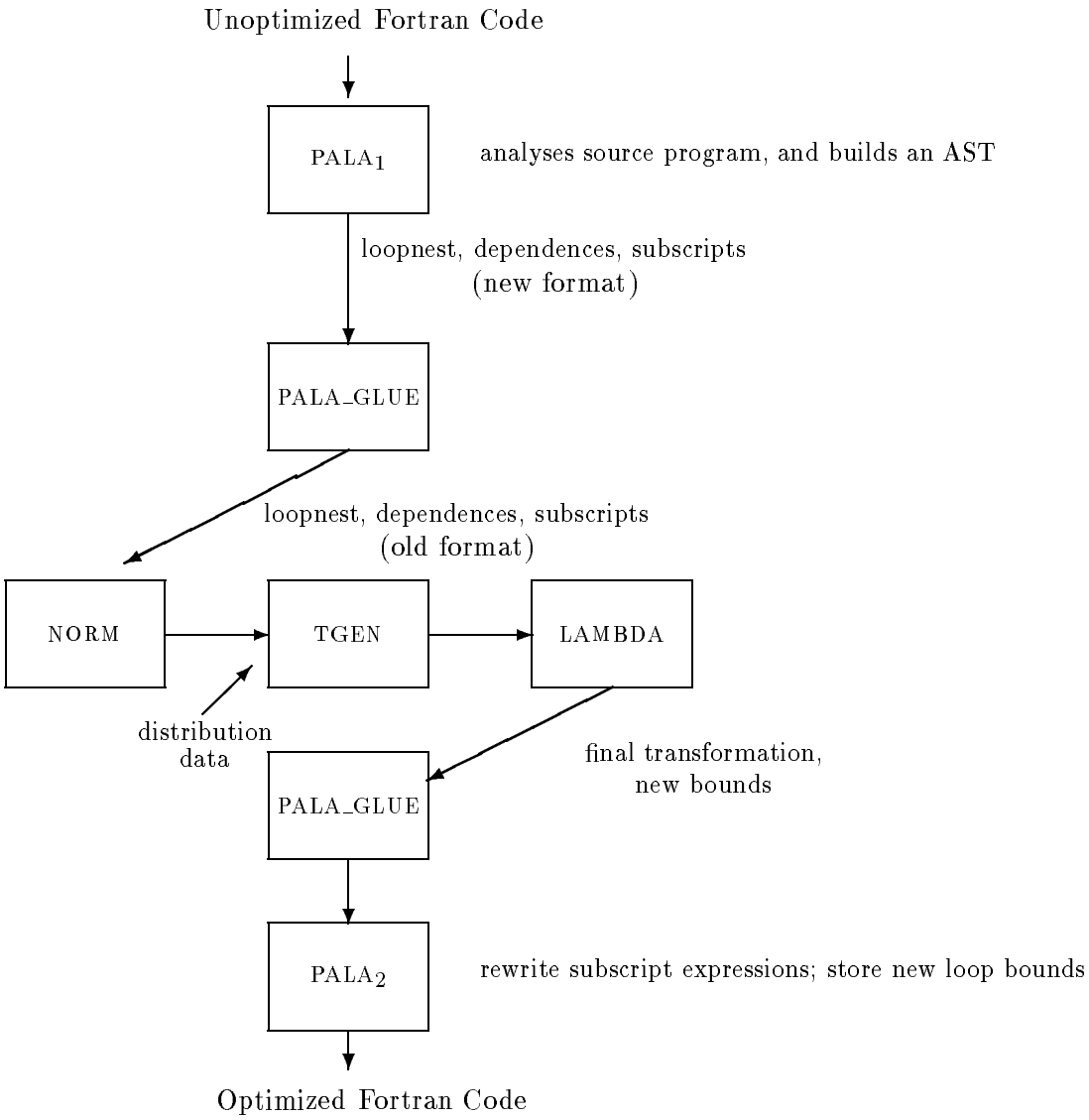

Figure 5: Actual Integration of ParaScope and Lambda. Because the interface specification of Lambda was changing during this project we decided to have Parascope (PALA) call the transformation routines in the new interface format, interposing a special translation program (PALA_GLUE) inbetween to translate between the old and new interfaces. It takes the data from ParaScope, allocates old-style data structures, and calls the lambda library in the old format. Upon return, it unpacks the old structures into the new interface and returns to ParaScope.

References

[1] Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[2] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, November 1989.

[3] Utpal Banerjee. Unimodular transformations of double loops. Technical Report Rpt. No. 1036, CSRD, August 1990.

[4] 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.

[5] 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.

[6] K. Fletcher, K. Kennedy, K. S. McKinley, and S. Warren. The parascope editor: User interface goals. Technical Report TR90-113, Department of Computer Science, Rice University, 1990.

[7] M. Hall, T. Harvey, K. Kennedy, N. McIntosh, K. McKinley, J. Oldham, M. Paleczny, and G. Roth. Experiences using the parascope editor: an interactive parallel programming tool. Technical Report CRPC-TR93297, Rice University, February 1992.

[8] K. Kennedy, K.S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[9] W. Li. *Lambda:* a loop transformation tool-kit (user's reference manual. May 1993.

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

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

[12] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. Technical Report CTC92TR98, Cornell Theory Center, July 1992.

[13] Applied Parallel Research. *FORGE 90 Baseline System User's Guide*. 550 Main Street, Suite I, Placerville, CA 95667, April 1992.

[14] Private communication.

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

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

APPENDIX A: Project Plan

We originally planned to proceed as follows:

1. Learn about Λ.
2. Figure out ParaScope's internal representation of loop bounds, loop body, dependences, array subscripts.
3. Do systems design for transcribing this data from ParaScope into Λ.
4. Figure out how to invoke Λ, and with what parameters if any.
5. Figure out how to transcribe Λ's output back into ParaScope.
6. Add the appropriate matrix-based transformation to ParaScope's Transformations menu.
7. Devise some test programs to exercise the Λ functionality.
8. Test the generated code by running the transformed Fortran emitted by ParaScope through FORGE 90 DMP (Distributed Memory Parallelizer) to create a message-passing program which can then be run on a network of IBM RS/6000 workstations.

We finished steps 1–5 and part of 7–8.

APPENDIX B: Parascope's Transformations