Generation of High Performance Domain-Specific Languages from Component Libraries

Ken Kennedy
Rice University

Collaborators

Raj Bandypadhyay
Zoran Budimlic
Arun Chauhan
Daniel Chavarria-Miranda
Keith Cooper
Jack Dongarra
Mary Fletcher
Rob Fowler

John Garvin
Lennart Johnsson
Chuck Koelbel
Cheryl McCosh
John Mellor-Crummey
Dan Sorensen
Linda Torczon
Anna Youssefi

http://www.cs.rice.edu/~ken/Presentations/TelescopeOSU.pdf

A Bit of History

- In the beginning, there was machine language (or “assembly” language)
- 1957: Fortran (John Backus,et.al., IBM)
  - made it possible for every scientist to develop (high-performance) applications
- Today: programming high end machines is once again the near-exclusive domain of experts.

Making Languages Usable

It was our belief that if FORTRAN, during its first months, were to translate any reasonable “scientific” source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

— John Backus

Philosophy

- Compiler Technology = Off-Line Processing
- Goal: Abstraction without guilt:
  - programming language usability without sacrificing performance
- Rule: performance of both compiler and application must be acceptable to the end user

Examples

- PL/I interpretive macro facility
  - Fixed macros can be compiled
    - 10x improvement with compilation
- TransMeta “Code Morphing”
  - Dynamic compilation of machine code

The Programming Problem

- Programming is complex, particularly on new platforms
- Professional programmers are in short supply (and expensive)
- Programming systems that result in low performance will not be accepted

Telescoping Languages

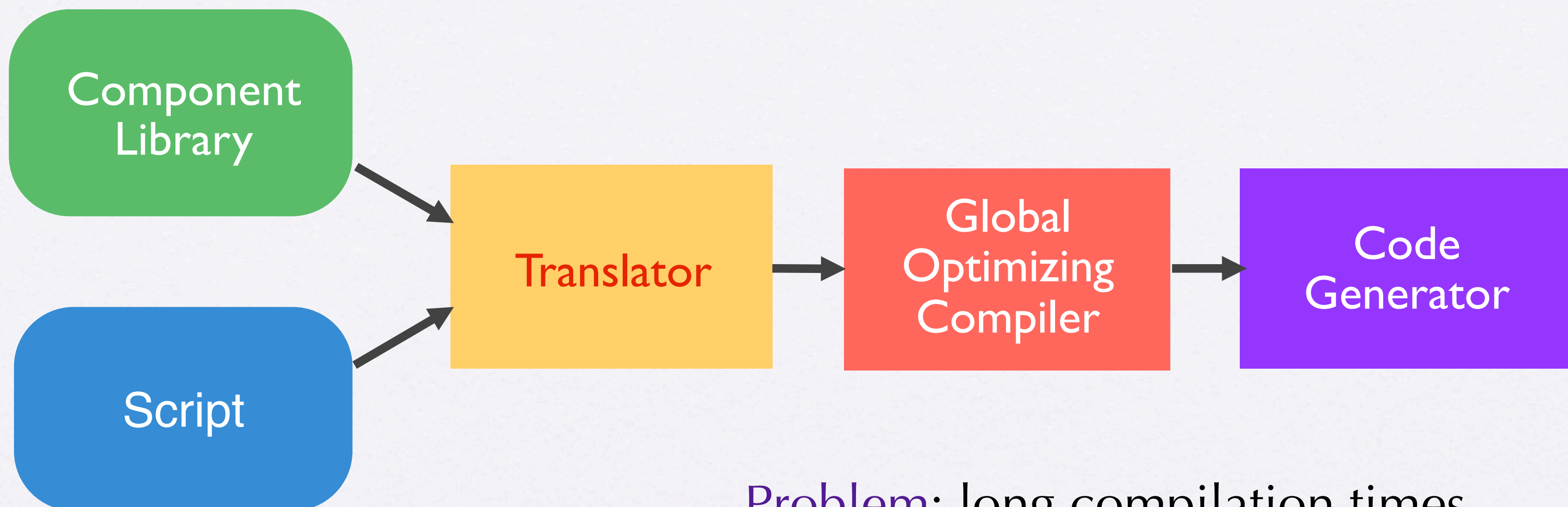
The Programming Problem: A Strategy

- Make it possible for end users to become application developers:
  - users integrate software components using *scripting languages* (e.g., Matlab, Python, Visual Basic, S+)
  - professional programmers develop software *components*

An Obstacle: Performance

- Conventional Strategy for Performance Improvement:
  - translate scripts and components to common intermediate language
  - optimize the resulting program using whole-program compilation

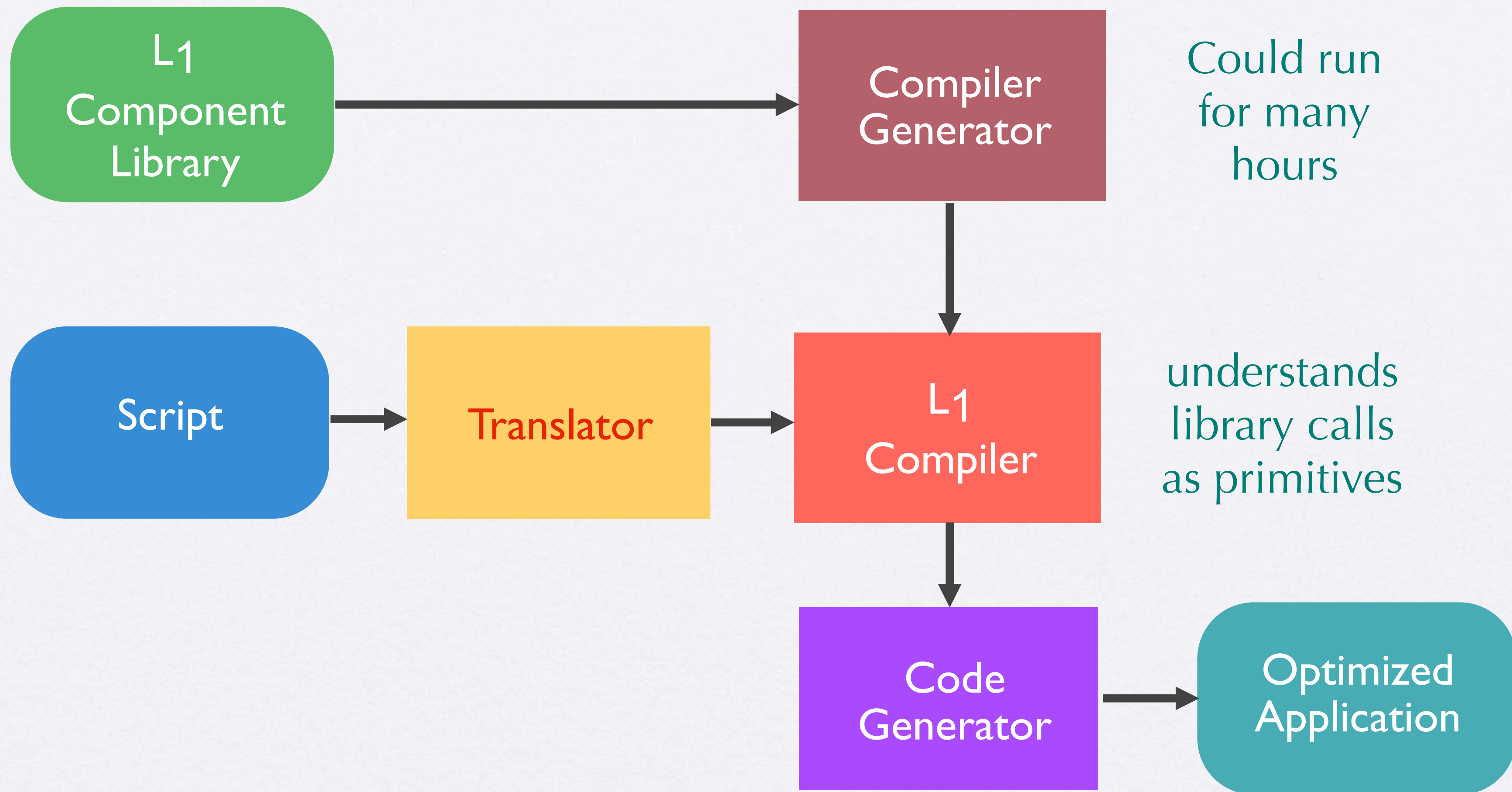
Whole-Program Compilation



Problem: long compilation times, even for short scripts!

Problem: expert knowledge on specialization lost

Telescoping Languages



Telescoping Languages: Advantages

- Compile times can be reasonable
- High-level optimizations can be included
- User retains substantive control over language performance
  - Generate a new language with user-produced libraries
- Reliability can be improved

Applications

Applications

- Matlab SP (Signal Processing)
  - Optimizations applied to functions
- Statistical Calculations in Science and Medicine in S (R or S-PLUS)
- Library Generation and Maintenance
- Component Integration Systems
- Parallelism in Matlab
- Scripting Grid Computations

Technologies

- Compilation of scripting languages
  - Matlab, R, Python, ...
  - Translation to C or Fortran
- Optimization of library accesses
  - Real focus of telescoping languages
  - Additional burden on compilation strategy

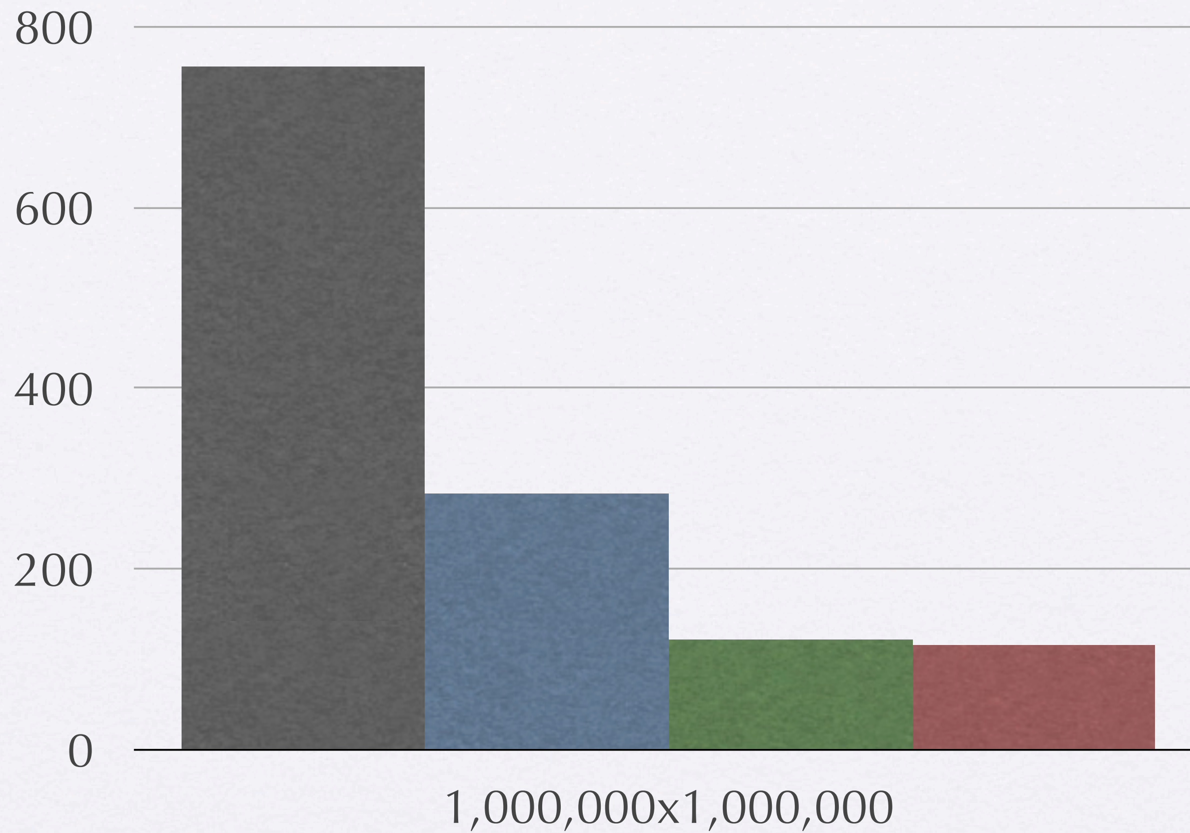
Matlab Compilation

- Type analysis: to determine variable type, and array rank, and size
  - Avoid reallocation in loops
- Vectorization: conversion of loops to array expressions
  - Exploit efficiency of LAPACK

Library Generator (LibGen)

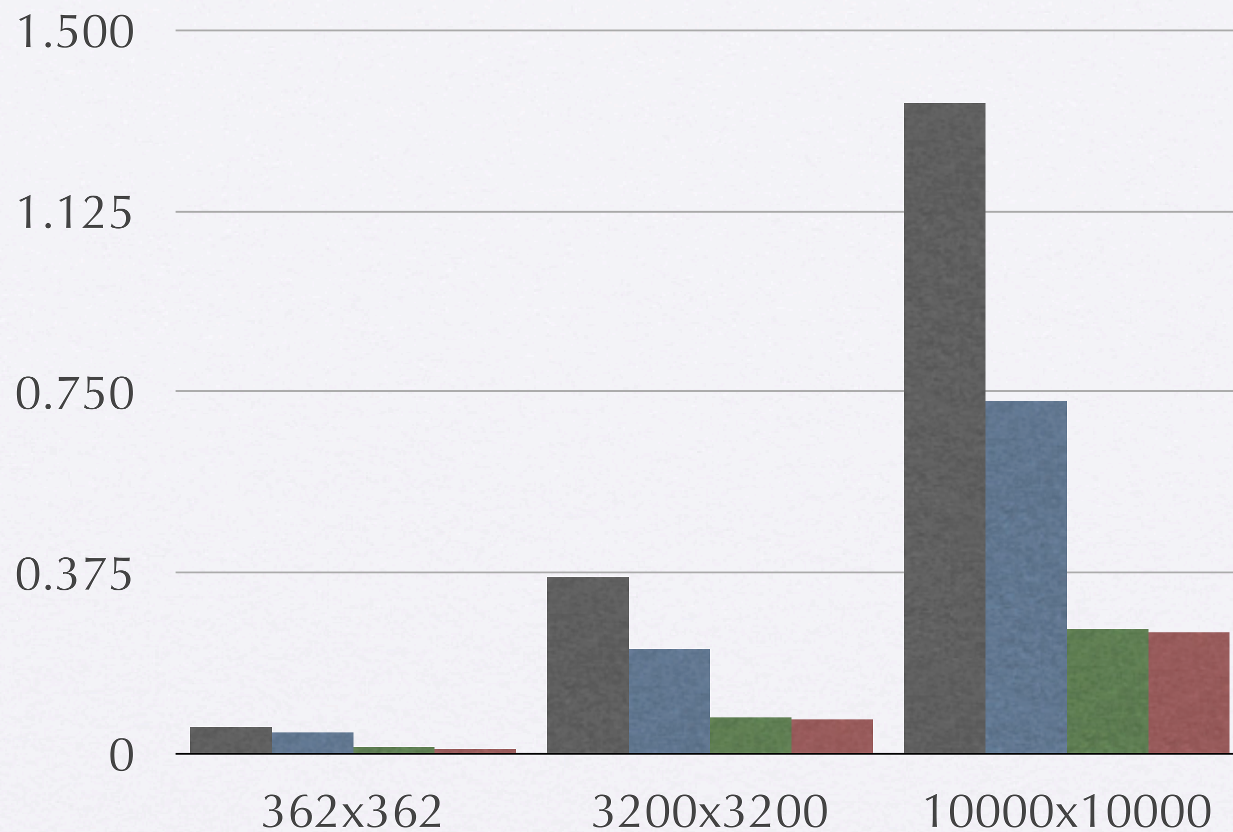
- Prof Dan Sorensen (Rice CAAM) maintains ARPACK, a large-scale eigenvalue solver
- He prototypes the algorithms in Matlab, then generates 8 variants in Fortran by hand:
  - ({Real, Complex} x {Single, Double} x {Symmetric, Nonsymmetric}
    - Special handling for {Dense, Sparse}
- Could this hand generation step be avoided?

LibGen Performance



LibGen Scaling

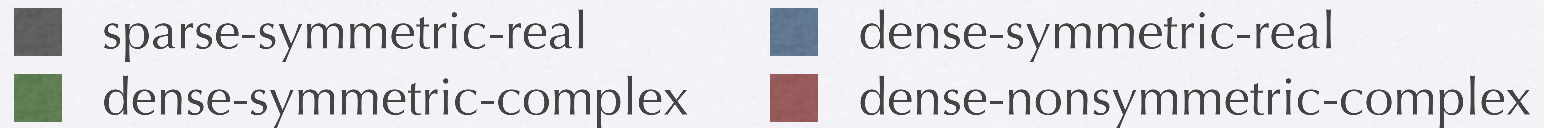
MATLAB 6.1 MATLAB 6.5 ARPACK LibGen

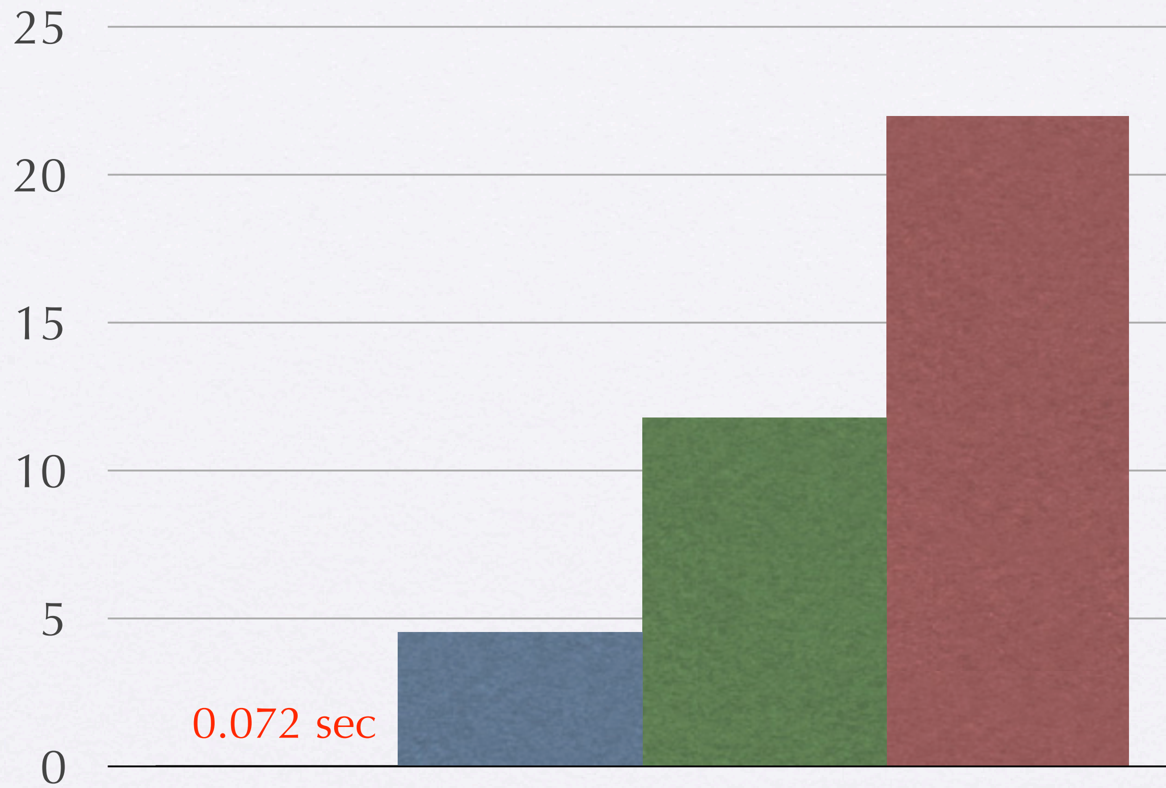


Key Technology: Type Inference

- Translation of Matlab to C or Fortran requires type inference
  - At each point in a function, translator must know types as functions of input parameter types ("type jump functions")
- Constraint-based type inference algorithm
  - Polynomial time
  - Can be used to produce different specialized variants based on input types

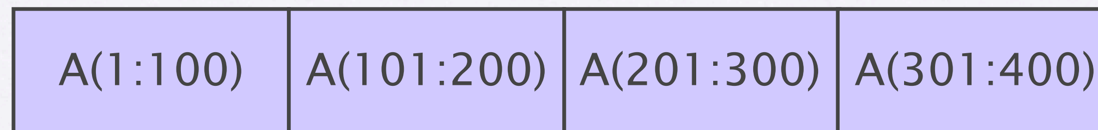
Value of Specialization





Parallelism in Matlab

- Add distributions to Matlab arrays
  - Distributions can cross multiple processors



- Use distributions to determine parallelism
  - Hide parallelism in library component operations
  - Specialize for specific distributions

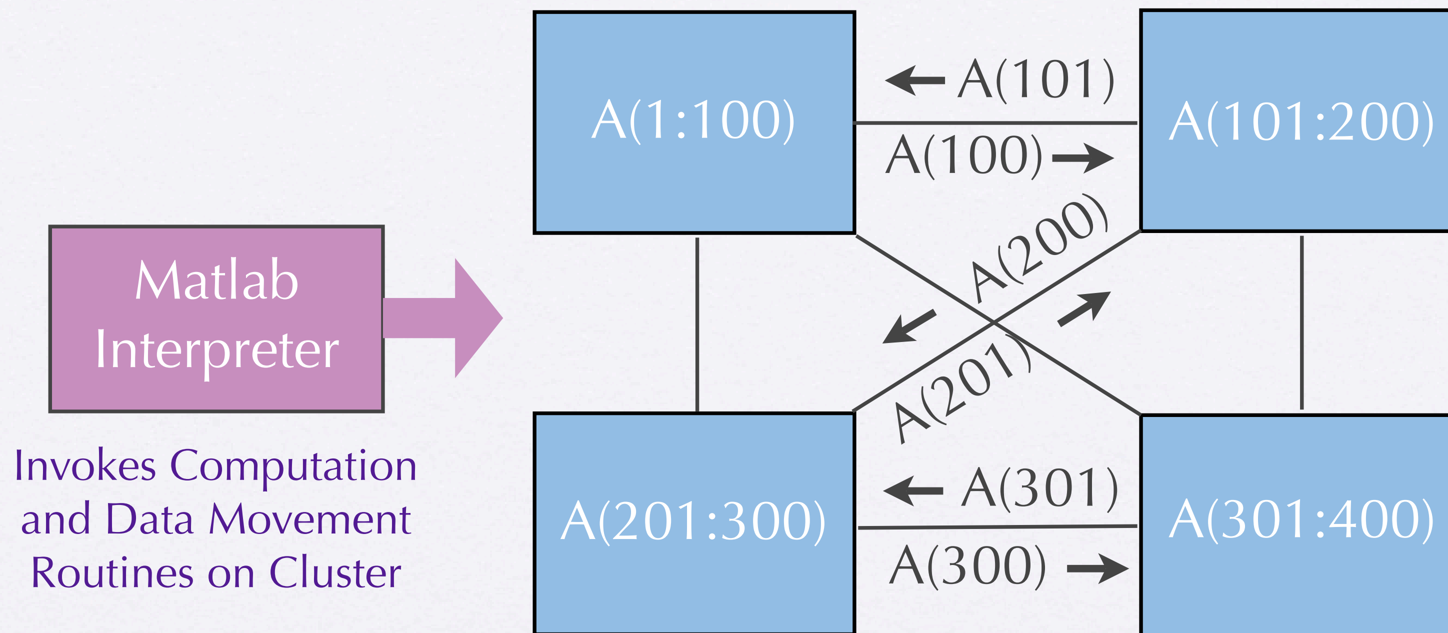
Parallel Matlab

```
D = distributed(n,m,block,block);

A = B + C .* D

A = D(1:n-2,1:m) + D(3:n,1:m)
```

Where is each operation performed?

Some data must be communicated!

Parallel Matlab Implementation



A(2:399) = (A(1:398) + A(3:400))/2

Parallel Matlab Implementation

- Produce parallel versions of functional components
  - for each supported distribution
- Produce data movement components
  - for each distribution pair

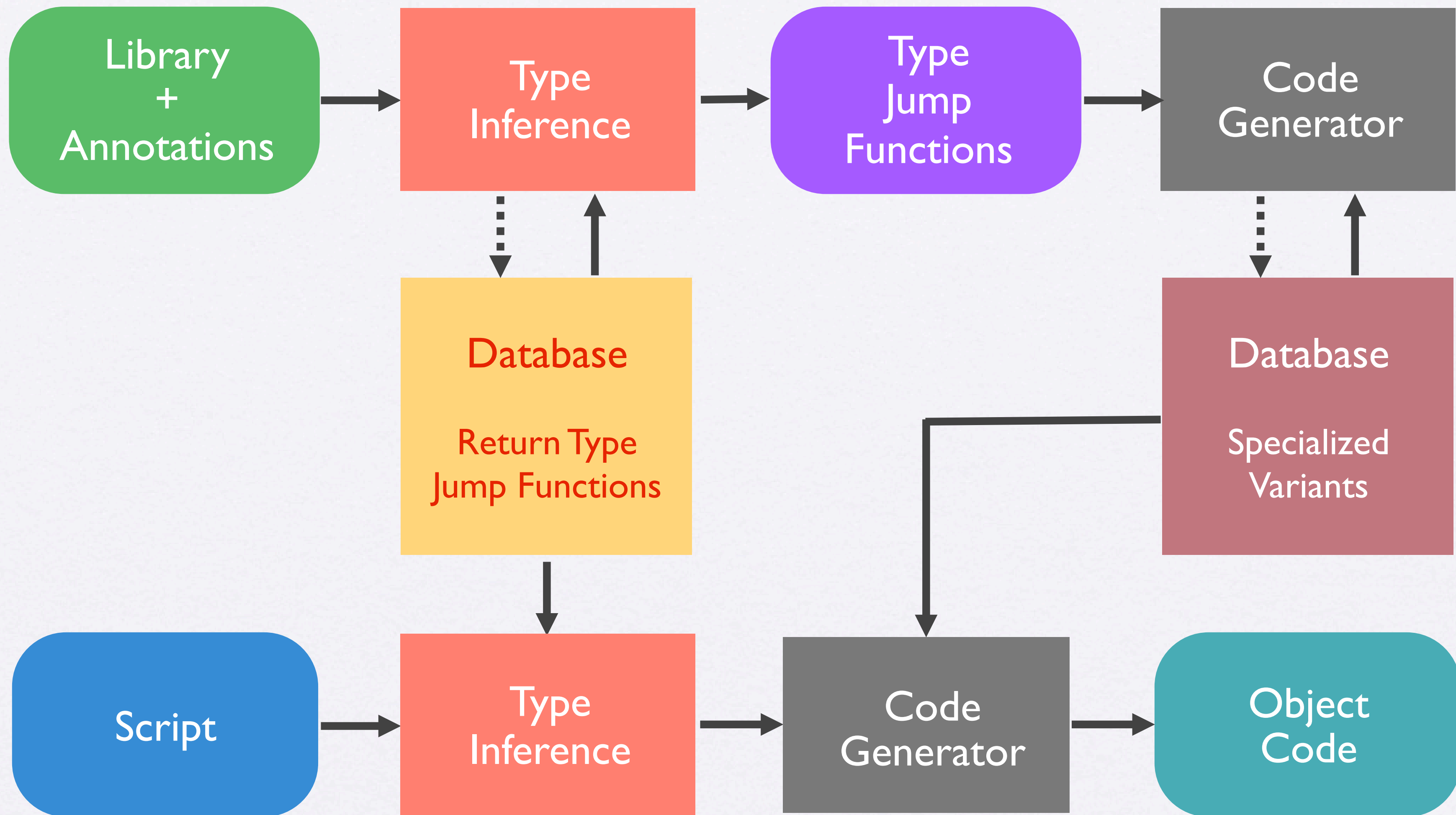
This sounds like a lot of work!

Specialization: HPF Legacy

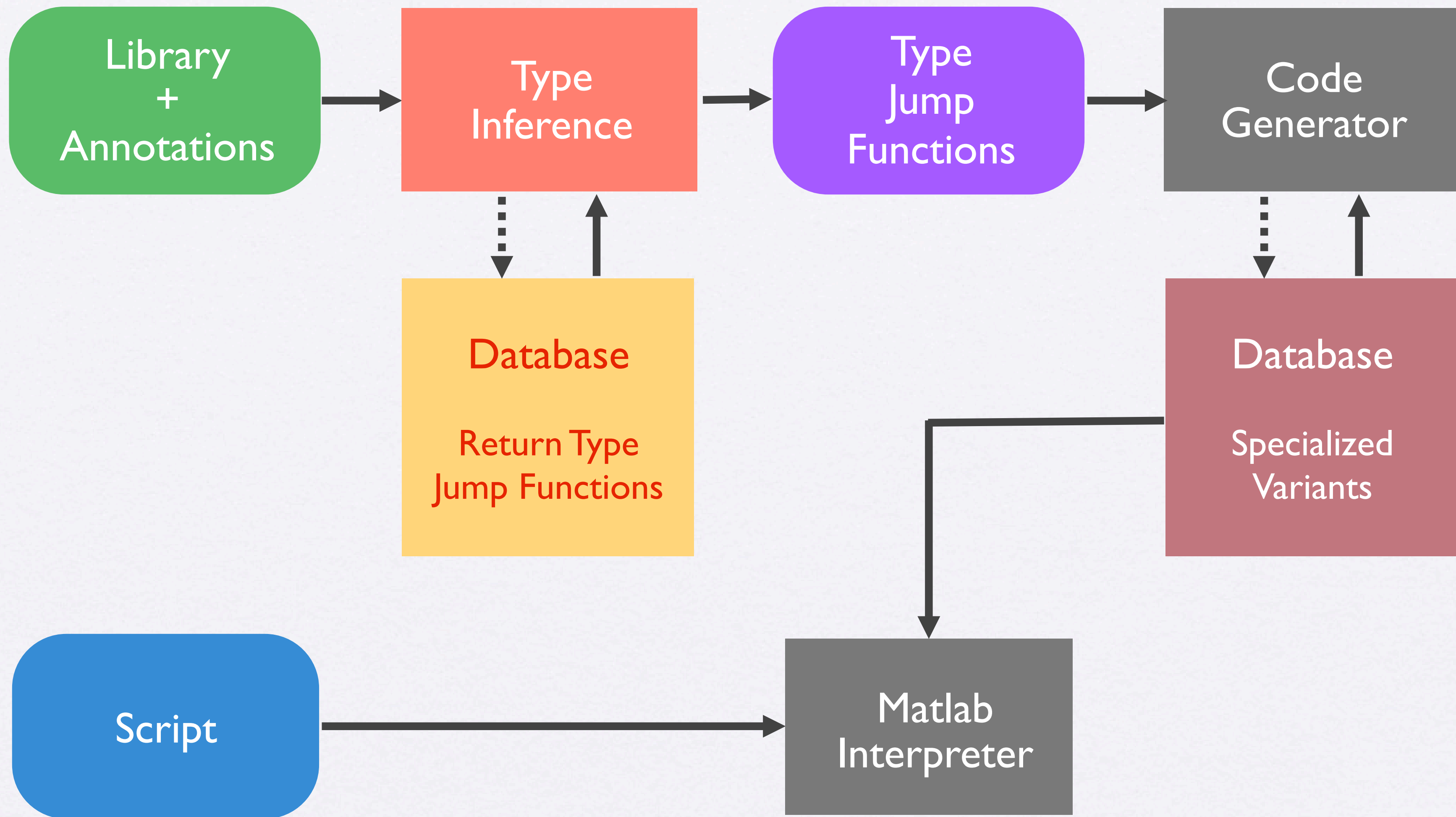
- Library written with template distribution
- Useful distributions specified by library designer (standard + user-defined)
  - Examples: multipartitioning, generalized block, adaptive mesh
- Library specialized using HPF compilation technology to exploit parallelism
  - Performed at language generation time

Implementation

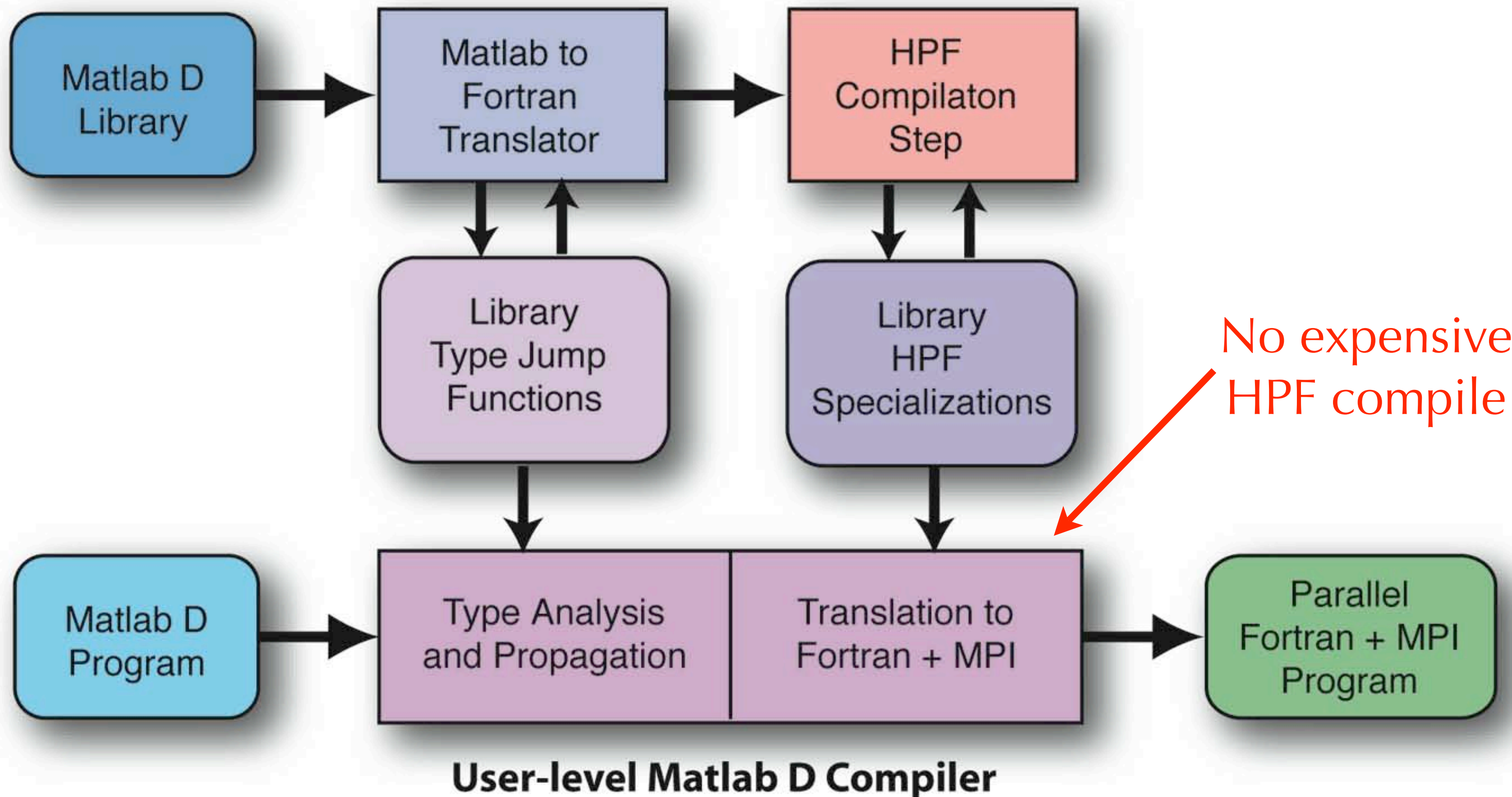
Base Matlab Implementation



JIT Implementation



Parallel Matlab



No expensive HPF compile

Aligning Distributions

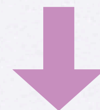
- Where to compute?

```
A = B + C .* D
```

on owner(A?)

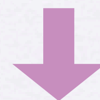
- How to align?

```
A = B(1:n-2) + C(3:n)
```



General

```
A = plus(coerce(B,dist(A),+1),
         coerce(C,dist(A),-1),
         dist(A))
```



Specialized

```
A = plus_shft(B,C,dist(A),+1,-1)
```

Summary

- Goal: make professionals, especially scientists and engineers, more productive
- Challenge: the need to balance developer productivity against application performance
- Strategy: explore technologies that directly translate prototyping languages to production code (even for parallel systems)

http://www.cs.rice.edu/~ken/Presentations/TelescopeUCSD.pdf

The End