CHAPEL + LAPACK
"NEW DOG, MEET OLD DOG."

Ian Bertolacci

INTRO: WHAT IS CHAPEL

Chapel is a high performance programming language that has been in development at Cray since 2005.

It includes many parallel programming language features, from data-parallelism, to task parallelism, to distributed computing.

Disclaimer: I worked for Cray Inc. with the Chapel team this summer (hence this talk), but the opinions are all mine.

INTRO: WHAT IS CHAPEL

```
var IterationSpace: domain(2) = {1..10,1..5};
var A, B, C : [IterationSpace] real;
for idx in A.domain do A[idx] = -idx : real;
forall idx in IterationSpace do B[idx] = 1/A[idx];
[ value in A ] value = value**2;
C = A + B;
var sum = + reduce C;

// learn more at: chapel.cray.com
//                learnxinyminutes.com/docs/chapel/
```

INTRO: WHAT IS LAPACK

Linear **A**lgebra **PACK**age.

Interface, not a specific set of code

- Often comes from vendors, and is highly optimized for their machines.
- Lots of groups use/implement/port LAPACK
  - Netlibs, PLAPACK, ScaLAPACK, MAGMA, PLASMA, MKL, LAPACK++, Nmath, Jblas, Matrix Toolkit Java, …

Related to **B**asic **L**inear **A**lgebra **S**ubprograms

- BLAS is used to implement LAPACK

WHY?

1. The success of a language lies on more than just the features of that language.
   - Libraries are an enormous draw for developers, and programmers will use whatever language the libraries can be easily interfaced with
2. (Potential) users had expressed that they needed LAPACK to be comfortable adopting Chapel as their language of choice.

GOALS

1. Create raw LAPACKE Chapel interface
   - ***Not a reimplementation/src-src transformation!***

CHAPEL INTERFACE

Target LAPACK**E** (Netlibs/Intel C interface)

- 3122 functions

Declare C functions as Chapel external procedures:

C declaration:

```
int LAPACKE_dgesv( int n, int nrhs, double* a, int lda, int* ipiv, double* b, int ldb );
```

Chapel declaration:

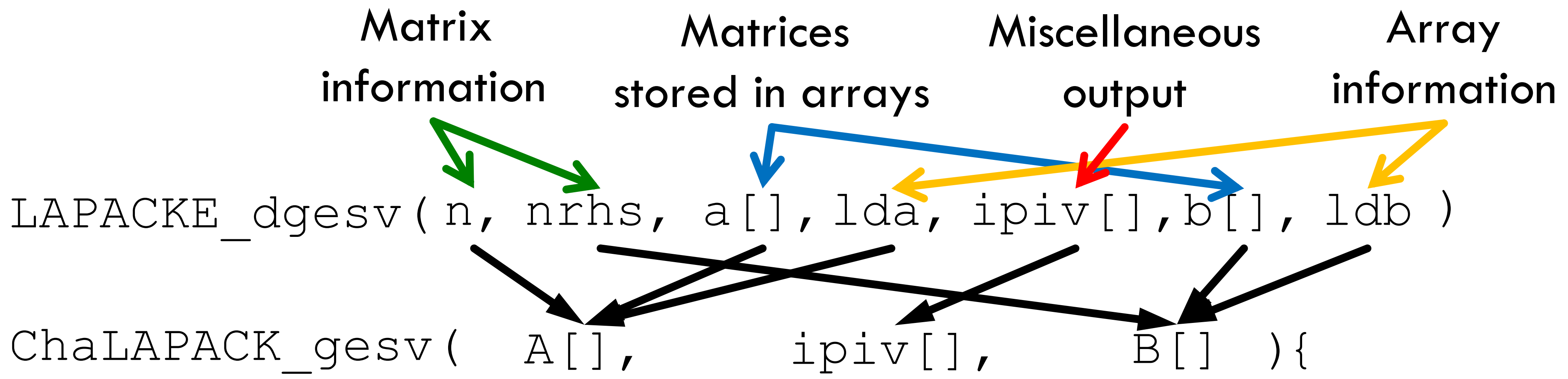
```
extern proc dgesv( n : c_int, nrhs : c_int, a : [] c_double, lda : c_int,
                   ipiv : [] c_int, b : [] c_double, ldb : c_int) : c_int;
```

GOALS

1. Create raw LAPACKE Chapel interface
   - ***Not a reimplementation/src-src transformation!***
2. Create Chapel abstractions.

CHAPEL ABSTRACTIONS

Many arguments could be abstracted by attributes of Chapel's arrays and domains.



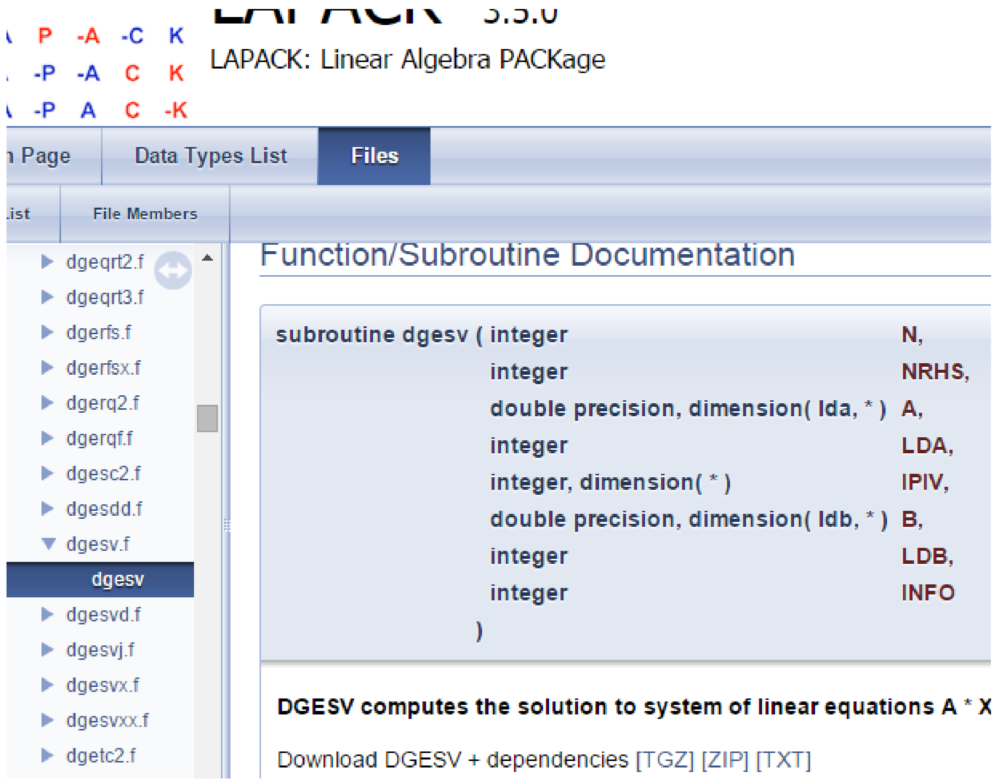
```
LAPACKE_dgesv( n, nrhs, a[], lda, ipiv[], b[], ldb )

ChaLAPACK_gesv(  A[],       ipiv[],      B[]  ){
  LAPACKE_dgesv( A.domain.dim(1).size, B.domain.dim(2).size,
                 A, A.domain.dim(2).size, ipiv, B,
                 B.domain.dim(2).size );
}
```

GOALS

1. Create raw LAPACKE Chapel interface
   - ***Not a reimplementation/src-src transformation!***
2. Create Chapel abstractions.
3. Generate Chapel documentation.

CHAPEL DOCUMENTATION

Utilize existing LAPACK documentation for chpldocs



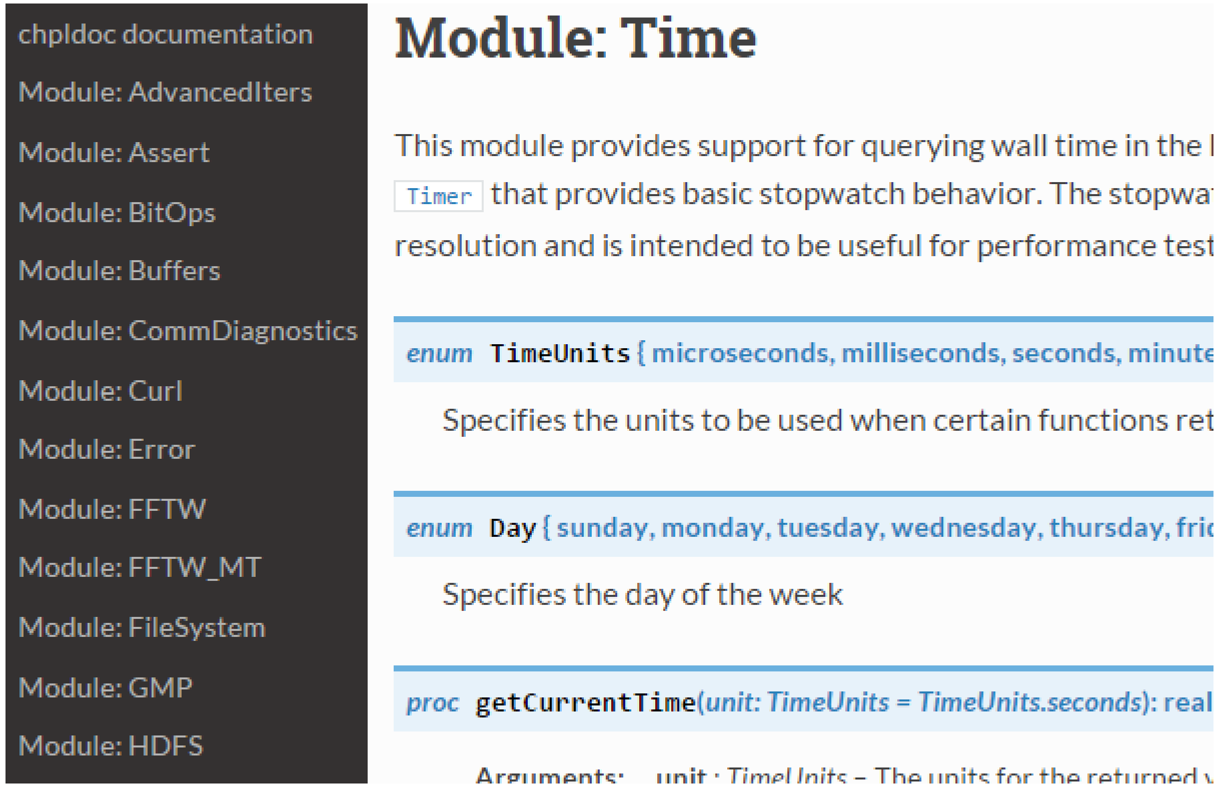
LAPACK: Linear Algebra PACKage

Function/Subroutine Documentation

```
subroutine dgesv ( integer                                  N,
                   integer                                  NRHS,
                   double precision, dimension( lda, * )    A,
                   integer                                  LDA,
                   integer, dimension( * )                  IPIV,
                   double precision, dimension( ldb, * )    B,
                   integer                                  LDB,
                   integer                                  INFO
                 )
```

DGESV computes the solution to system of linear equations A * X

Download DGESV + dependencies [TGZ] [ZIP] [TXT]



Module: Time

This module provides support for querying wall time in the l
`Timer` that provides basic stopwatch behavior. The stopwa
resolution and is intended to be useful for performance test

enum `TimeUnits` { microseconds, milliseconds, seconds, minute

Specifies the units to be used when certain functions ret

enum `Day` { sunday, monday, tuesday, wednesday, thursday, fri

Specifies the day of the week

proc `getCurrentTime`(*unit: TimeUnits = TimeUnits.seconds*): *real*

Arguments: unit : *TimeUnits* – The units for the returned v

GOALS

1. Create raw LAPACKE Chapel interface
   - ***Not a reimplementation/src-src transformation!***
2. Create Chapel abstractions.
3. Generate Chapel documentation.
4. Column major array mapping.

COLUMN MAJOR ARRAYS

Many LAPACK implementations are written in Fortran, where column major ordered in memory

Chapel, C, and many other languages are row major order.

Converting from one to the other wastes cycles.

GOALS

1. Create raw LAPACKE Chapel interface
   - ***Not a reimplementation/src-src transformation!***
2. Create Chapel abstractions.
3. Generate Chapel documentation.
4. Column major array mapping.

SUCCESSES

1. Full coverage of LAPACKE.
2. Generated functions to abstract many of the original arguments.
3. Too big. Also tripped on red tape.
4. We didn’t get around to it.

EASY RIGHT?

Nope.

WHY I DISLIKE C

```
foo( int* a )
```

What is `a`? Or rather, what's at `a`?

- Scalar?
- Array?
  - How big?
  - Into… start, end, middle?
  - Single dimension? Or Flat-packed multidimensional array?
    - Column or Row major?

Repeat after me: C is a machine semantics language.

WHAT DOES LAPACK ACTUALLY LOOK LIKE?

Actual LAPACKE function declaration:

```
lapack_int LAPACKE_dgesvd(
        int matrix_order,char jobu, char jobvt,
        lapack_int m, lapack_int n, double* a,
        lapack_int lda, double* s, double* u,
        lapack_int ldu, double* vt, lapack_int ldvt,
        double* superb );
```

Pointer to 1D array

Pointers to 2D arrays

Pointer to scalar

LETS TRY SEMANTIC ANALYSIS!

No, thank you.

Hard

- I tried to build ROSE and it took two weeks for me to give up.

Not guaranteed

- `*(ptr+i)` is the same as `ptr[i]` : False Negative
- `ptr[0]` is the same as `*ptr` : False positive

DOCUMENTATION ANALYSIS

LAPACK is quite well documented; though occasionally broken (written by humans).

Arguments are documented with:

- Type (double, int, complex)
- Arrangement type (scalar, array)
- Intent (in, out, inout)
- Human description of what they are for ("n is the number of rows in matrix a")

Most importantly, it can be searched through with regex.

However, we are still targeting LAPACKE; the documentation lives in LAPACK's Fortran source code.

PROCESS: TYPES

1. Collect all LAPACKE and LAPACK functions, associate their symbols, and toss any LAPACK functions that don't exist in LAPACKE.
2. Fold type and arrangement type of LAPACK arguments onto their LAPACKE associates.
   - This identifies anonymous pointer arguments as scalars or arrays.
   - Array arguments also contribute the size(s) of the master array (important later).
3. If there are any arguments or functions in LAPACKE that have not been properly resolved: figure it out by actually reading the documentation (with eyes) and enter information by hand
   - Script creates a list of what it thinks should be there, and you verify (often quick)
   - It happens, but often small enough to be accomplished in under an hour
4. Generate extern procedure declarations. (Satisfy goal #1)

DOCUMENTATION ANALYSIS: MEANING

To create the Chapel abstracted functions, and remove arguments from the procedures, it is necessary to know what those arguments mean.

The human readable documentation for each argument can be searched through with a convoluted regular expression to find key terms (such as ‘rows’, ‘columns’) and the name of the other arguments that they related to

PROCESS: MEANING

1. Search the arguments documentation for key terms, and associate arguments.
2. Bind attribute arguments to components of their associate arguments.
3. Remove bound arguments from procedure signature.
4. Generate new function with truncated signature.
5. Fill body of function with call to original function, where each callee argument is either a pass through of an argument from the caller, or an attribute of an argument from the caller. (Satisfies goal #2)

CONCLUSION

- Using a rigid method of searching documentation text, it was possible to create a Chapel interface to LAPACK via the LAPACKE C interface.
- Such methods may be viable for larger and more complex codes, and may become stables of automated program analysis
- Program analysis is crucial to advance mature, important software onto new hardware, paradigms, and languages.