Computational Methods in Astrophysics

Dr Rob Thacker (AT319E)

thacker@ap.smu.ca

Today's Lecture

HPC Libraries

- Part 1: Common libraries + opportunities for linking to other languages, especially Python

- Good exercises in R to try (with solutions)
- https://bcgts.files.wordpress.com/2016/12/rexercises.pdf

Shifts in the programming Model..

"We need to move away from a coding style suited for serial machines, where every macrostep of an algorithm needs to be thought about and explicitly coded, to a higher-level style, where the compiler and library tools take care of the details. And the remarkable thing is, if we adopt this higher-level approach right now, even on today's machines, we will see immediate benefits in our productivity."

W. H. Press and S. A. Teukolsky, 1997
Numerical Recipes: Does This Paradigm Have a future?

Motivations, concerns

- In developing large applications, three significant issues must be addressed:
  - Productivity
    - Time to the first solution (prototype) and time to solution (production)
  - Complexity
    - Increasingly sophisticated models, may need to link to other solvers
  - Performance
    - Increasingly complex algorithms, architectures
- What strategies should be applied?
  - Some appear mutually exclusive: best performance would reduce productivity if you tailor every single part of the code

Unavoidable tension

Scientists frequently need highest performance

Algorithms have long lifetimes (longer than hardware)

Low level programming

High level programming

Library approach

- Why not use libraries? (provided they suit your problem)
  - Optimization – many library functions are often assembly optimized
  - Well tested – libraries are used by far more people than your local research group
  - Support – frequently commercial packages come with online forums or email support
- Main drawback – loss of understanding of code inner workings
  - Is this really an issue? 99.9% of the software you use you didn't write
  - Also you are forced into using the library interface, but usual this is not a significant concern
- Secondary drawback may be cost

Library ownership

- Three main possibilities
  - Public Domain
    - Most common for numerical software
  - Commercial
    - Uncertain where this will go – Universities under pressure to gain income from licencing
  - Vendor Specific
    - Many of the big vendors release platform specific optimized versions of the larger public domain packages

Potential benefits of libraries

- Allows easier collaboration (provided library is freely available to everyone!)
- Software using GPL'd libraries can be released publicly as source-code
  - You can contribute back improvements to the user community
- Source based libraries can be adapted to your needs
- Bottomline is that your time to solution is reduced!

Libraries go mainstream

- Although libs have been around for 40 years, there's been an explosion in usage in the past 5-10 years
- Sociological changes in science + growth of GPL have made code sharing almost de riguer
- Python arguably leads the way, but it was somewhat necessary for the language to make a big impact
  - Numpy – arrays + a bit of linear algebra
  - Scipy – variety of useful ODE solvers, functions, constants, genetic algorithms + plenty more
  - https://wiki.python.org/moin/NumericAndScientific

Notable Public Domain Numerical Libraries (1970s(!)-today)

- LAPACK
  - Linear equations, eigenproblems (in R, Scipy)
- BLAS
  - Fast linear algebra kernels (in R, Scipy)
- LINPACK
  - Linear equation solving (now incorporated in LAPACK)
- ODEPACK
  - Ordinary d.e. solving (see also Scipy, odespy)
- QUADPACK
  - Numerical Quadrature (Scipy interface plus C version in GSL)
- ITPACK
  - Sparse problem

Basic Linear Algebra Subprograms (BLAS)

- **FORTRAN** library of simple subroutine which can be used to build more sophisticated LA programs (dates back to 1970's)
- **BLAS** is divided into four types and three levels
  - Single, double, complex and double complex
  - Level 1 (vector-vector operations) (order n)
  - Level 2 (matrix-vector operations) (order n^2)
  - Level 3 (matrix-matrix operations) (order n^3)
- Functions are prefixed with the type of the variables:
  - s,d,c, or z for **s**ingle, **d**ouble, **c**omplex, or double complex (z).

BLAS routines

- Some of the BLAS 1 subprograms are:
  - xCOPY - copy one vector to another
  - xSWAP - swap two vectors
  - xSCAL - scale a vector by a constant
  - xAXPY - add a multiple of one vector to another
  - xDOT - inner product
  - xASUM - 1-norm of a vector
  - xNRM2 - 2-norm of a vector
  - IxAMAX - find maximal entry in a vector

Levels 2 & 3

- Some of the BLAS 2 subprograms are:
  - xGEMV - general matrix-vector multiplication
  - xGER - general rank-1 update
  - xSYR2 - symmetric rank-2 update
  - xTRSV - solve a triangular system of equations
- Some of the BLAS 3 subprograms are:
  - xGEMM - general matrix-matrix multiplication
  - xSYMM - symmetric matrix-matrix multiplication
  - xSYRK - symmetric rank-k update
  - xSYR2K - symmetric rank-2k update

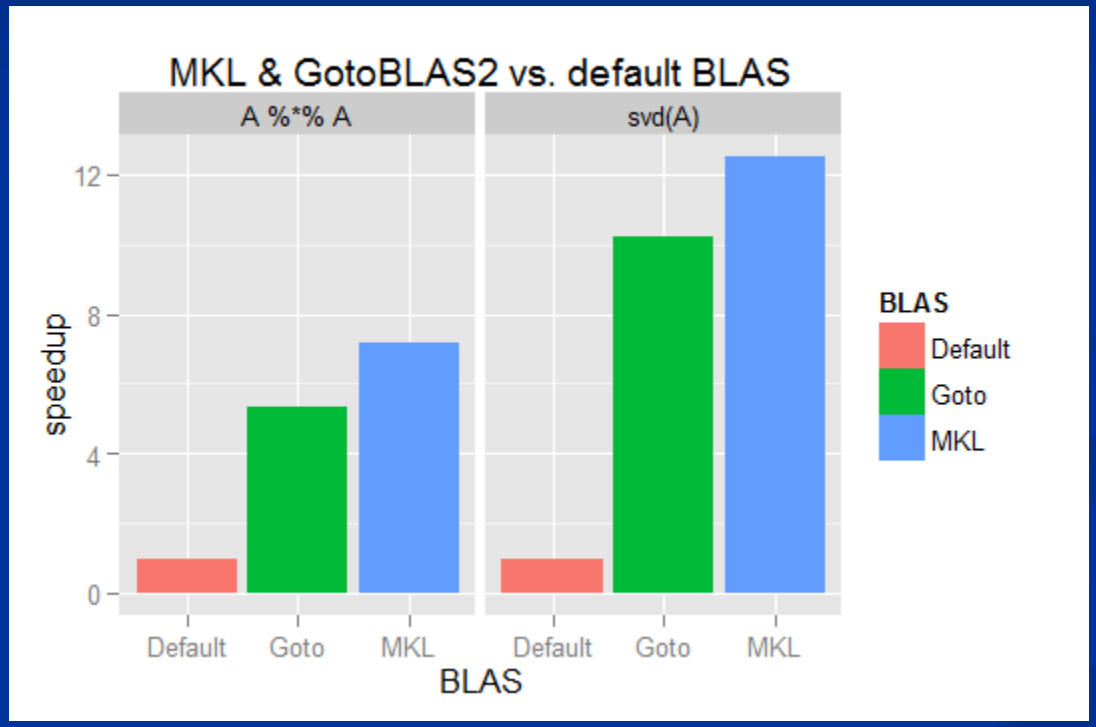
BLAS and C

- CBLAS is a C version of the libraries
  - Available from Netlib
- However, you can still call FORTRAN versions from C
  - you will need to declare the involved BLAS routine as “extern”
  - ```
    extern void dgemv_(char *trans, int *m, int *n,
    double *alpha, double *a, int *lda, double *x, int
    *incx, double *beta, double *y, int *incy );
    ```
    ```

BLAS in R

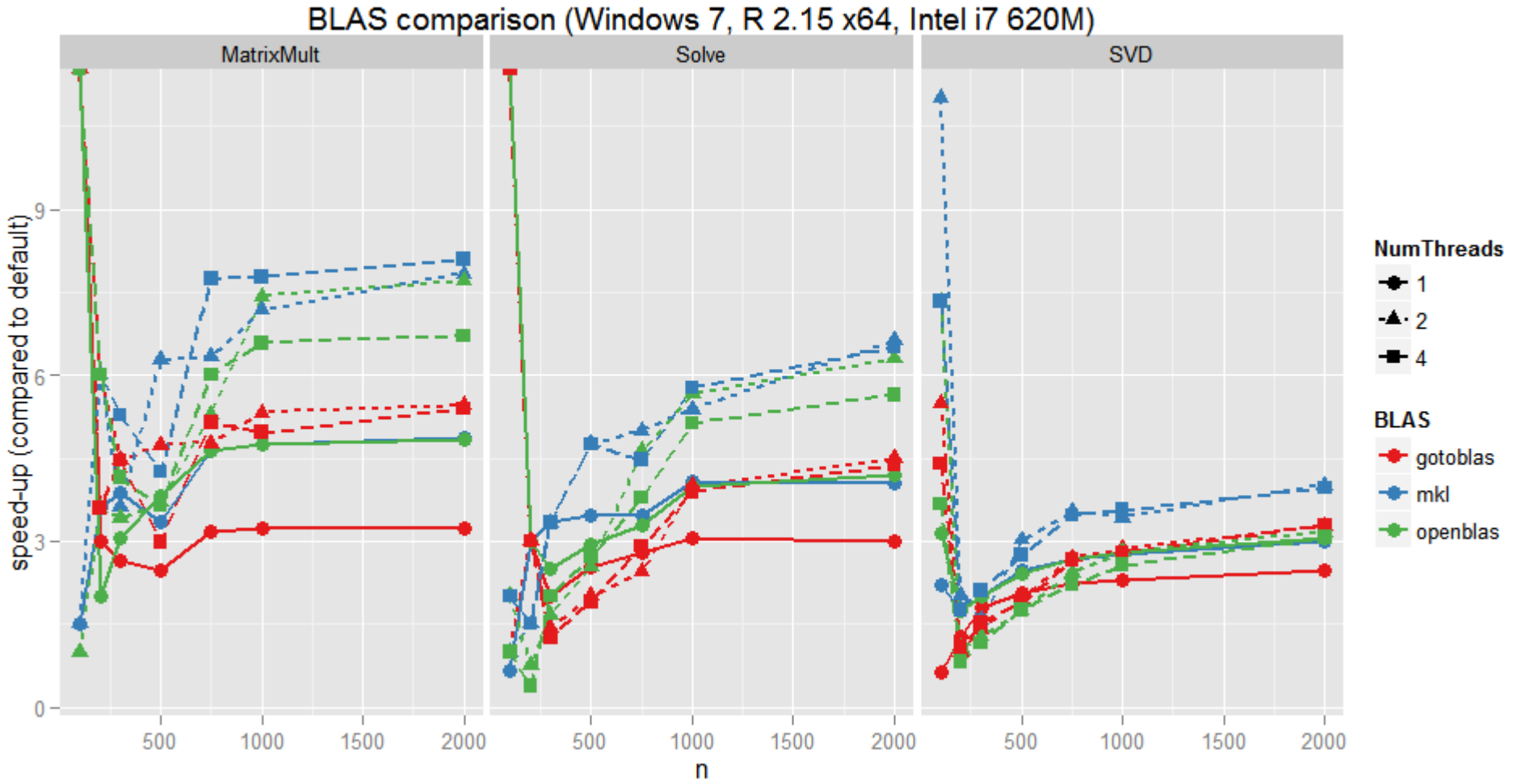
- R's native BLAS is not optimal but you can recompile with more optimized libs such as
  - ATLAS
  - ACML (AMD)
  - Intel MKL
  - OpenBLAS
- See for example https://csantill.github.io/RPerformanceWBLAS/
- Generally see vast speed up switching from native to ATLAS or OpenBLAS (operation dependent - factors of 8 possible)
- MKL can be even faster, but comes at a cost for non-academic users (several hundred dollars)
  - If you are a student you can get it free – great!
  - If not, you're most of the way there is you use ATLAS or OpenBLAS

Regular BLAS vs MKL in R



From http://blog.felixriedel.com/2012/10/speed-up-r-by-using-a-different-blas-implementation/
See https://brettklamer.com/diversions/statistical/faster-blas-in-r/ for more recent (2021) data

OpenBLAS vs MKL in R



From http://www.r-bloggers.com/r-blas-gotoblas2-vs-openblas-vs-mkl/

LAPACK

- **BLAS** is used as the building block for the **L**inear **A**lgegra **Pack**age, **LAPACK**

- Website describing and distributing a portable version of the library: http://www.netlib.org/lapack/
  - Includes online manual
    - http://www.netlib.org/lapack/lug/index.html
  - Vendors frequently distribute their own assembly level optimized versions of the library (e.g. Intel MKL, and AMD AOCL)

- This library consists of a set of higher level linear algebra functions with interface described at:
  - http://www.netlib.org/lapack/individualroutines.html

Goals of LAPACK

- Originally developed for shared memory and vector machines with a goal to combining LINPACK and EISPACK
  - Even shared memory machines have non-trivial memory hierarchies that need to be taken into account
- LAPACK relies upon block-matrix operations to improve data reuse and avoid long waits for data
  - Essentially optimizations made for challenging architectures usually help a little on less complicated architectures
  - The result is a "transportable" library
- Internally it calls BLAS routines
  - You can thus link in you favourite optimized BLAS library

LAPACK

- There are a very large number of linear algebra subroutines available in **LAPACK**
  - All follow a XYYZZZ format, where X denotes the datatype, YY the type of matrix and ZZZ describes the computation performed. For example:
    - **dgetrf** is used to compute LU factorizations of a matrix (d=double, ge=general, trf=triangular factorization)
    - **dgetrs** uses an LU factorization from dgetrf to solve a system
    - **dgetri** uses the LU above to compute the inverse of a matrix
    - **dgesv** essentially a combined call to dgetrf and dgetrs
    - **dgeev** computes the eigenvalues of a matrix.

Gnu Scientific Library

- http://www.gnu.org/software/gsl/
- GSL is a numerical library for C and C++ programmers
- Free software, available under GNU GPL
- PyGSL also available: http://pygsl.sourceforge.net
- The library provides a wide range of mathematical routines
  - There are over 1000 functions in total.
- The project was conceived in 1996 by Dr M. Galassi and Dr J. Theiler of Los Alamos National Laboratory.

GSL Features

- The library uses an object-oriented design
  - Different algorithms can be plugged-in easily or changed at run-time without recompiling the program
- It is intended for ordinary scientific users
  - Users with a knowledge of C programming will be able to use the library quickly
- Interface is designed to be simple to link into very high-level languages, such as Python
- Library is thread-safe
- Many of the routines are C "re"implementations of FORTRAN routines (e.g. FFTPACK)
  - Modern coding conventions and optimizations have been applied

Full list of functions

Complex Numbers

Vectors and Matrices

BLAS Support

Fast Fourier Transforms

Quasi-Random Sequences

Histograms

Simulated Annealing

Numerical Differentiation

Discrete Hankel Transforms

Least-Squares Fitting

Roots of Polynomials

Permutations

Linear Algebra

Quadrature

Random Distributions

N-Tuples

Differential Equations

Chebyshev Approximation

Root-Finding

Physical Constants

Special Functions

Sorting

Eigensystems

Random Numbers

Statistics

Monte Carlo Integration

Interpolation

Series Acceleration

Minimization

IEEE Floating-Point

Compiling and Linking

- The library header files are installed in their own `gsl' directory
  - Include statements need `gsl/' directory prefix:

    #include <gsl/gsl_math.h>
- Compile objects first: gcc -c myprog.c
- Then link: gcc example.o -lgsl -lgslcblas -lm

FFTW

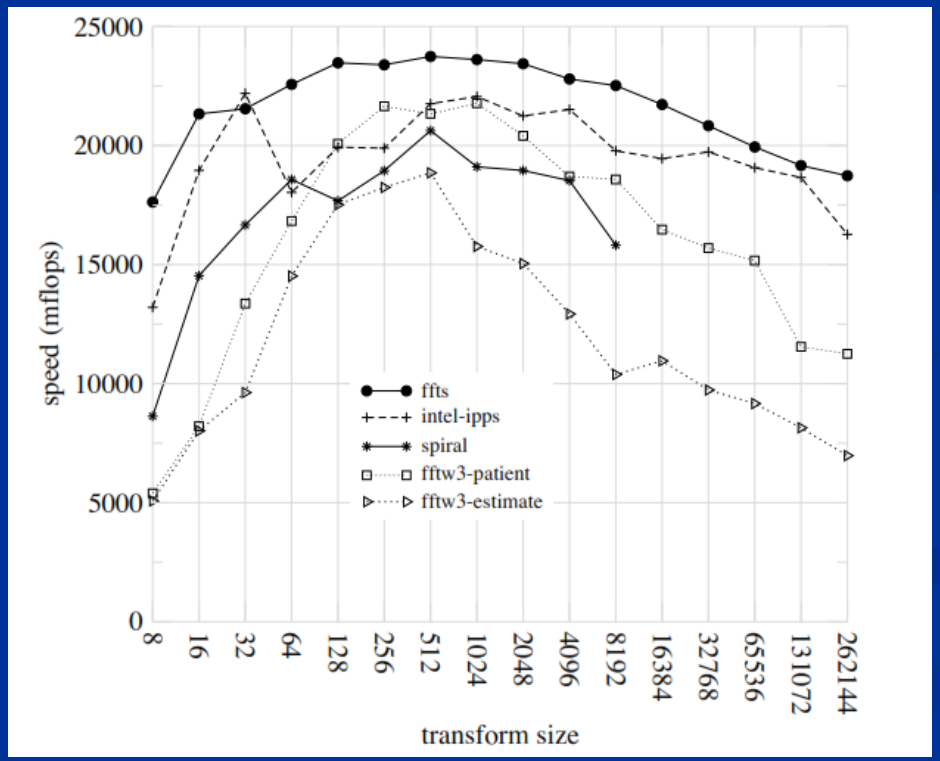
- http://www.fftw.org
- "Fastest Fourier Transform in the West"
- Authored by Frigo and Johnson at MIT
- C subroutine library for discrete Fourier transforms
  - Portable
  - Multiple dimensions
  - Arbitrary input sizes, real and complex transforms
    - Small prime factors are best though
  - Discrete cosine and sine transforms
  - Parallel versions available (both shared (pthreads) and distributed memory (MPI))
  - C and FORTRAN API
  - Supports SIMD extensions (e.g. SSE)
- *Self-tuning*
  - Contains many different FFT algorithms and optimal one is chosen at runtime
- Has undergone a number of evolutions, and is now at version 3.3.4
- Won 1999 J. H. Wilkinson Prize for Numerical Software

Using FFTW

- Need to include header files
  - `#include <fftw3.h>` or `include "fftw3.f"`
- Must also link to libraries
  - `-lfftw3 -lm` but may also need to specify path – will be installation dependent
- Having created arrays("in" and "out"), must create a "plan"
  - *`plan=fftw_plan_dft_1d(N,in,out,FFTW_FORWARD,FFTW_ESTIMATE)`*
  - *`call dfftw_plan_dft_1d(plan,N,in,out,FFTW_FORWARD, FFTW_ESTIMATE)`*
  - Precise plan routine will depend upon the FFT operation you wish to perform
  - Call to plan allows system to evaluate architecture and transform and then optimize the algorithm to be used in the FFT
- Having created the plan the transform is executed by specifying fftw_execute(plan)
- See the fftw website for precise details, plus pyFFTW is available

FFTS

- Prior to entering private sector NZ PhD student Anthony Blake developed a new self-tuning approach
  - Dubbed "Fastest Fourier Transform in the South"
- Available on github: https://github.com/anthonix/ffts
- For a number off FFT configurations it is faster than FFTW, although not for all



Core i7-2600 single precision power of 2 ffts

Parallel FFTW: Shared Memory

- FFTW includes both a pthreads based SMP library and can be compiled with OpenMP support on platforms where it is available
- Threaded version requires additional memory
  - Call `fftw_init_threads()` before using the threaded version
- SMP parallel plans require knowledge of how many threads are going to be used
  - Call `fftw_plan_with_nthreads(nthreads)`
  - Note that since plans are specific to the number of threads, if you change the number of threads you must create a new plan
- When work is completed you must call `fftw_cleanup_threads()` deallocate memory for threads
- At linking stage must also include parallel library
  - `-lfftw3_threads`
- Note only `fftw_execute` is using a parallel region

Parallel FFTW: MPI

- Now available on new v 3.x libraries
- MPI data decomposition is "slab" based.
  - For 3d arrays this is potentially limiting – can only use L processors if you have an L^3 array
  - However, communication costs are high so this is not often a significant barrier
- Uses MPI_Alltoall primitive which can occasionally lead to poor performance (depends on MPI implementation)
- Must enable support when FFTW is compiled and must also link to `-lfftw_mpi`

Example code

```
#include <fftw_mpi.h>

int main(int argc, char **argv)
{
      const int NX = ..., NY = ...;
      fftwnd_mpi_plan plan;
      fftw_complex *data;

      MPI_Init(&argc,&argv);

      plan = fftw2d_mpi_create_plan(MPI_COMM_WORLD,
                                    NX, NY,
                                    FFTW_FORWARD, FFTW_ESTIMATE);

      ...allocate and initialize data...

      fftwnd_mpi(p, 1, data, NULL, FFTW_NORMAL_ORDER);

      ...

      fftwnd_mpi_destroy_plan(plan);
      MPI_Finalize();
}
```

AMD Optimizing CPU Libraries (AOCL)

- AMD had a proprietary "Core Math Library" ACML for years, it was useful but not as extensive as Intel's
- Now using an open source distribution with slightly different focus
- https://developer.amd.com/amd-aocl/
- EPYC CPU optimized code
  - BLAS + LAPACK-type functions
  - FFTW
  - Random number generators

Intel oneAPI Math Kernel Library

- Version 2021.4 recently released
- Free use in academic environment
- Online support forum
- Threads supported
- Library functions:
  - Linear Algebra - BLAS and LAPACK
  - Linear Algebra - PARDISO Sparse Solver
  - Discrete Fourier Transforms
  - Vector Math Library
  - Vector Statistical Library
    - random number generators
- Also have MPI ScaLAPACK implementation

Random Number Generators

- Numerical recipes RAN2 and RAN3 are both reasonable RNGs – but there are better options
  - Note RAN3 does fail some of the more esoteric tests
- GSL library provides over 40 different generators
  - Includes Knuth's algorithms
  - Mersenne Twister as well
- Numpy's rng is Mersenne twister – but make sure you set the seed yourself

Mersenne Twister

- http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
- Developed by Matsumoto and Nishimura
- Period is 2^19937-1 (10^{6000})
  - 623-dimensional equidistribution property is assured
- Moderate speed – some consider it slow though
- Efficient use of the memory
  - The implemented C-code mt19937.c consumes only 624 words of working area
- Currently the generator of choice for most problems
  - except crypto – given 624 pieces of the sequence one can predict all following numbers
- There's a "tiny" version that uses less bits, but has shorter period

Need for speed

- Often many RNs are need
  - e.g. Monte Carlo simulation
- Saito & Matsumoto developed a SIMD-oriented version of the Mersenne Twister
  - SFMT
- Roughly twice as fast as the most optimized version of MT, 3-4x fast than non-optimized MT
- http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/

Crypto-secure RNG

- Guessable seeds are an issue
- Systems that rely on computer state can be potentially "hacked" by sending input
  - It looks like this can actually done carefully enough to avoid external hacks though
- Best CSRNGs adopt a true random process, e.g. radioactive decay, lava rand
- Google has released code for a CSRNG - https://code.google.com/archive/p/csrng/

Summary Part 1

- Libraries offer a number of benefits
  - Optimization
  - Robustness
  - Portability
  - Time to solution improvements