To code or not to code? Part II

Week 11

Optimization in C++

C++ optimization

- The previous optimizations were for all languages
- Now we will discuss about C++-specific optimizations
  - Inlining (already known)
  - Template meta programs
  - Lazy Evaluation
  - Expression templates

The C++ compiler : a Turing machine

- Erwin Unruh has shown that a C++ compiler with templates is a Turing machine
  - It can perform any arbitrary calculation that a classical computer can perform
  - The halting problem is undecidable: it is impossible to decide whether a C++ compiler will ever finish compiling a program
- In particular we can
  - Perform loops at compile time
  - Do branches at compile time
    - Using partial template specialization
- This is called “template meta programming”

Unruh's famous prime number program: unruh.C

- The following program prints all prime numbers as error messages

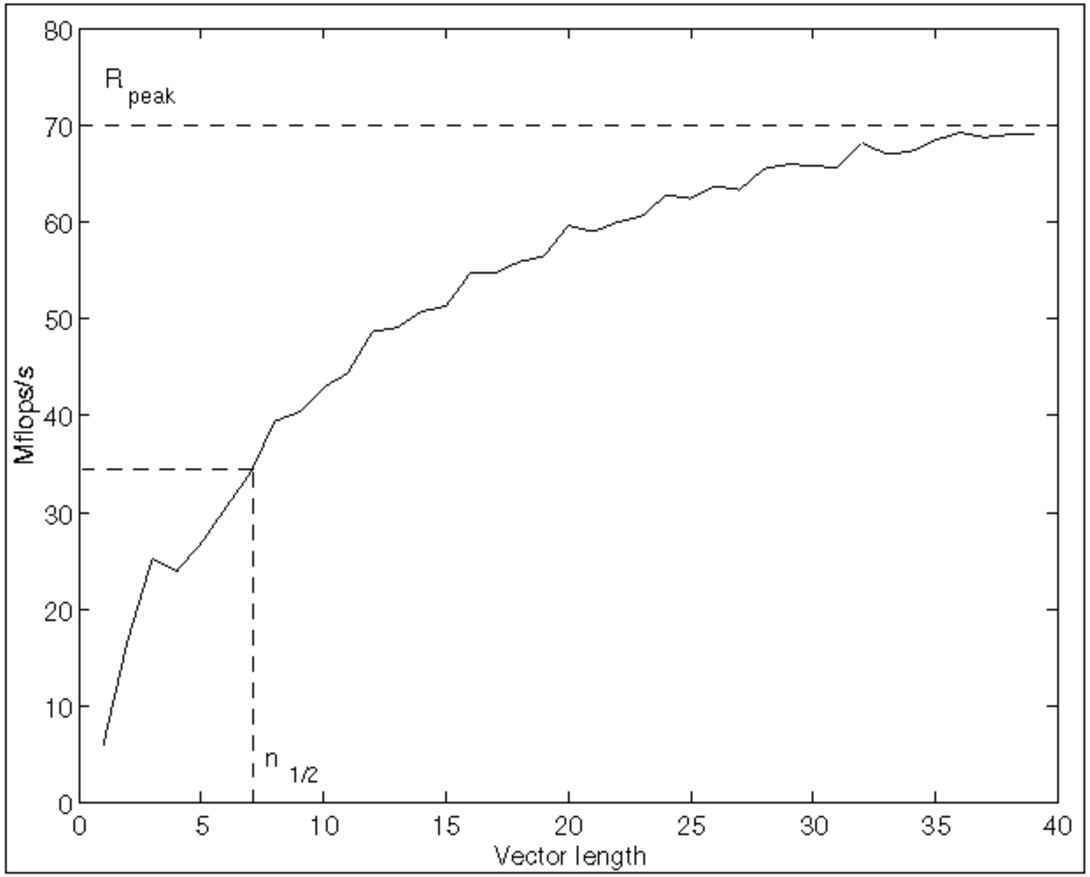
```
// Erwin Unruh, untitled program, ANSI X3J16-94-0075/ISO WG21-462, 1994.
template<int i> struct D { D(void*); operator int(); };

template<int p, int i> struct is_prime {
  enum { prim = (p%i) && is_prime<(i > 2 ? p : 0), i-1>::prim };
};

template<int i> struct Prime_print {
  Prime_print<i-1> a;
  enum { prim = is_prime<i,i-1>::prim };
  void f() { D<i> d = prim; }
};

struct is_prime<0,0> { enum { prim = 1 }; };
struct is_prime<0,1> { enum { prim = 1 }; };
struct Prime_print<2> {
   enum { prim = 1 };
   void f() { D<2> d = prim; }
};

void foo()
{ Prime_print<10> a; }
```

Performance bottleneck in dot products

- The dot product

```
double dot(const double* a, const double* b, int N)
{
  double result = 0.;
  for (int i=0; i < N; ++i)
    result += a[i] * b[i];
  return result;
}
```

does not reach peak performance on small vectors because the loop overhead is too large



(figure © by Todd Veldhuizen)

Unrolled loops reach peak performance

- The unrolled loop is optimal:

```
inline double dot3(const double* a, const double* b) {
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}
```

- Question: how can we unroll for vectors of size N, where N is a template parameter such as in `tinyvector.h` ?

```
template<class T, int N>
class TinyVector {
public:
  T& operator[](int i) { return data[i]; }
  T operator[](int i) const { return data[i]; }

private:
  T data[N];
};
```

- Answer: template meta programming

An unrolled dot product by template meta programming

```
// The dot() function invokes meta_dot -- the metaprogram
template<class T, int N>
inline float dot(TinyVector<T,N>& a,
  TinyVector<T,N>& b)
{ return meta_dot<N-1>::f(a,b); }

// The metaprogram
template<int I>
struct meta_dot {
  template<class T, int N>
  static T f(TinyVector<T,N>& a, TinyVector<T,N>& b)
  { return a[I]*b[I] + meta_dot<I-1>::f(a,b); }
};

template<>  // the end of the recursion
struct meta_dot<0> {
  template<class T, int N>
  static T f(TinyVector<T,N>& a, TinyVector<T,N>& b)
  { return a[0]*b[0]; }
};
```

How does this work?

- Here is what the compiler does with the code:

```
TinyVector<double,4> a, b;
double r = dot(a,b);

         = meta_dot<3>::f(a,b);

         = a[3]*b[3] + meta_dot<2>::f(a,b);

         = a[3]*b[3] + a[2]*b[2] + meta_dot<1>::f(a,b);

         = a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + meta_dot<0>::f(a,b);

         = a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + a[0]*b[0];
```

Operator overloading on a vector class: simplevector.h

- Let’s start with a simple vector:

```
template <class T>
class simplevector {
public:
  simplevector(int s=0);
  simplevector(const simplevector&
    v);
  ~simplevector()

  operator=(const simplevector& v)
  operator +=(const simplevector& v)

  int size() const
  T operator[](int i) const;
  T& operator[](size_type i);

private:
  value_type* p_;
  size_type sz_;
};
```

- And add an operator+

```
template <class T>
simplevector<T> operator+(
   const simplevector<T>& x,
   const simplevector<T>& y )
{
  simplevector<T> result=x;
  result += y;
  return result;
}
```

Why operator overloading is slow: `simplevector.h`

- Problem: The expression

  ```
  A=B+C+D
  ```

- for `std::valarray` or for the `simplevector` class is evaluated as

  ```
  TEMP1  = B+C;
  TEMP2  = TEMP1+D;
  A      = TEMP2;
  ```

  needs two extra write and two extra read operations. Time the programs `timesimple.C` and `timesimple2.C` for a first benchmark.

- Expression templates, developed by Todd Veldhuizen solve this. The expression is evaluated just as:

  ```
  for (int j=0;j<A.size();++j)
    A[j] = B[j] + C[j] + D[j];
  ```

Lazy evaluation: lazyvector.h and timelazy2.C

- Consider

```
A = B + C;
```

- We define a vectorsum object:

```
template <class T>
class vectorsum {
public:
  vectorsum(const lazyvector<T>& x,
    const lazyvector<T>& y)
  : left_(x), right_(y) {}

  T operator[](int i) const
  { return left_[i] + right_[i]; }

private:
  const lazyvector<T>& left_;
  const lazyvector<T>& right_;
};
```

- operator+ just returns the vectorsum object describing the sum:

```
template <class T>
inline vectorsum<T> operator+(
   const lazyvector<T>& x,
   const lazyvector<T>& y)
{ return vectorsum<T>(x,y); }
```

- The evaluation is only done when assigning, there is no temporary

```
template <class T>
class lazyvector {
  …
  operator=(const vectorsum<T>& v) {
    for (int i=0;i<size();++i)
      p_[i]=v[i];
  }
  …
};
```

Making it more flexible: `lazyvectorflex.h`

- We would need to write a class for each type of expression:
  - `vectorsum`
  - `vectordifference`
  - `vectorproduct`
  - …
- Instead let us define operation classes:

```
struct plus {
  template <class T> static inline T
    apply(T a, T b)
    { return a+b;}
};

struct minus {
 template <class T> static inline T
     apply(T a, T b)
     { return a-b;}
};
```

- template vectorsum on the operation:

```
template <class T, class Op>
class vectorsum {
public:
  vectorsum(const lazyvector<T>& x,
    const lazyvector<T>& y)
  : left_(x), right_(y) {}

  T operator[](int i) const
  { return Op::apply(
          left_[i],right_[i]);
  }

private:
  const lazyvector<T>& left_;
  const lazyvector<T>& right_;
};
```

Making it more flexible: lazyvectorflex.h

- Define operator+ and operator- by just this one class:

```
template <class T>
inline vectorsum<T,plus> operator
   +(
   const lazyvector<T>& x,
   const lazyvector<T>& y)
{ return vectorsum<T,plus>(x,y);

template <class T>
inline vectorsum<T,minus>
    operator-(
   const lazyvector<T>& x,
   const lazyvector<T>& y)
{ return vectorsum<T,minus>(x,y);
```

- And add the template also to the assignment:

```
template <class T>
class lazyvector {
  …
  template <class Op>
  operator=(const vectorsum<T,Op>& v)
    {
    for (int i=0;i<size();++i)
      p_[i]=v[i];
   }
   …
};
```

Expression templates

- Lazy evaluation solves the problem for expressions with only two operands:
  - `A = B + C`
  - `A = B - C`
  - `A = B * C`
  - `A = B / C`
- How can we make this more general, to allow also expressions such as:
  - `A = B + C + D`
  - `A = B * ( C + D ) + exp(E)`
- The answer is: templates! Here called “expression templates”

Step 1: generalizing vectorsum into an expression class

- Let us replace

```
template <class T, class Op>
class vectorsum {
public:
  vectorsum(const lazyvector<T>& x,
    const lazyvector<T>& y)
  : left_(x), right_(y) {}

  T operator[](int i) const
  { return Op::apply(
          left_[i],right_[i]);
  }

private:
  const lazyvector<T>& left_;
  const lazyvector<T>& right_;
};
```

- By a more general class

```
template <class L, class Op, class R>
class X {
public:
  X(const L& x, const R& y)
  : left_(x), right_(y) {}


  T operator[](int i) const
  { return Op::apply(
          left_[i],right_[i]);
  }

private:
  const L& left_;
  const R& right_;
};
```

Step 2: generalizing the operators

- Replace

```
template <class T>
inline vectorsum<T,plus> operator+(const lazyvector<T>& x,
                                   const lazyvector<T>& y)
{
  return vectorsum<T,plus>(x,y);
}
```

- By the more general

```
template <class L, class T>
inline X<L,plus,etvector<T> > operator+(const L& x,
                                        const etvector<T>& y)
{
  return X<L,plus,etvector<T> >(x,y);
}
```

Step 3: generalizing the assignment

- Replace

```
template <class T>
class lazyvector {
  …
  template <class Op>
  operator=(const vectorsum<T,Op>& v) {
    for (int i=0;i<size();++i)
      p_[i]=v[i];
   }
};
```

- by

```
template <class T>
class etvector {
  …
  template <class L, class Op, class R>
  operator=(const X<L,Op,R>& v) {
    for (int i=0;i<size();++i)
      p_[i]=v[i];
   }
};
```

How does this work

- First the compiler parses the expression:

```
D = A + B + C;
  = X<etvector,plus,etvector>(A,B) + C;
  = X<X<etvector,plus,etvector>,plus,etvector >
           (X<etvector,plus,etvector>(A,B),C);
```

- Then it matches the `operator=`:

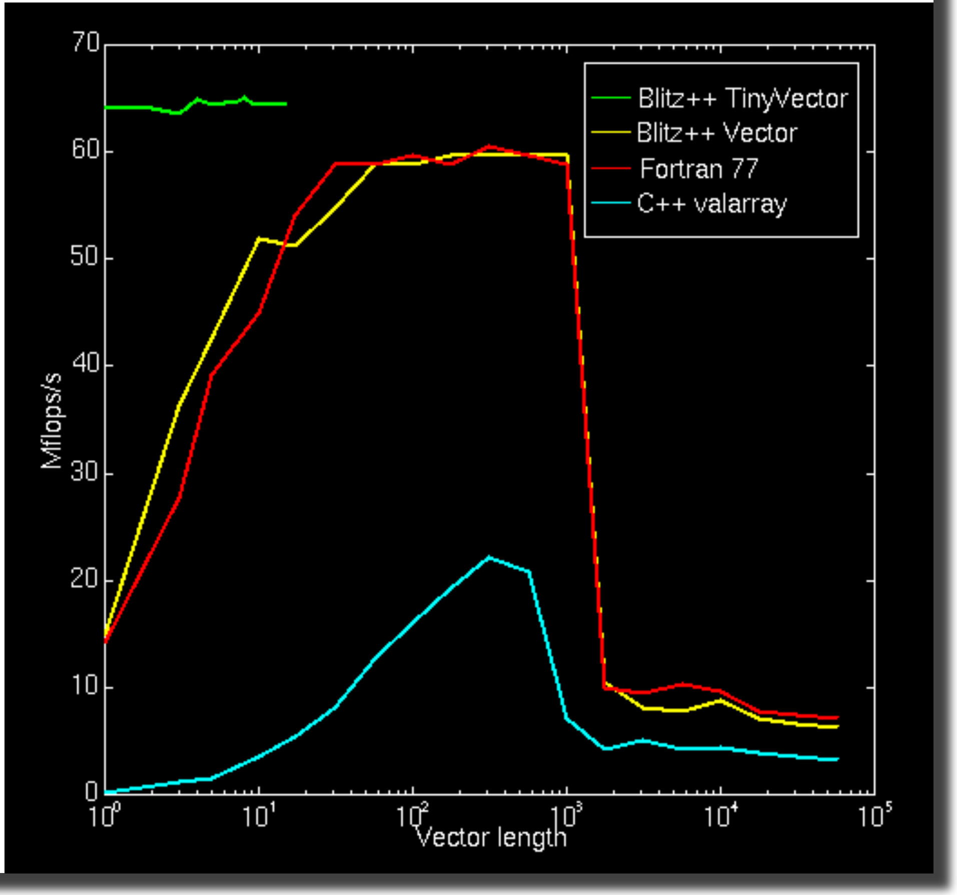
```
D.operator=(X<X<etvector,plus,etvector>,plus,etvector>
            (X< etvector,plus,etvector >(A,B),C) v) {
  for (int i=0; i < sz_; ++i)
    p_[i] = v[i];
}
```

- And for each index the expression is evaluated:

```
p_[i] = plus::apply(X<etvector,plus,etvector >(A,B)[i], C[i]);
       = plus::apply(A[i],B[i]) + C[i];
        = A[i] + B[i] + C[i];
```

Performance tests

- On the object orienten numerics page http://oonumerics.org/ you can find benchmarks of expression templates compared to Fortran
- C++ with expression templates
  - Has more than 90% of the performance of Fortran 77
  - Is faster than Fortran 90/95 or C



C++ libraries

- Fortran libraries are well tested and optimized but hard to use
- C++ libraries are rare but are being developed more intensely now
  - Blitz++: expression templates for array operations and stencils
  - POOMA: parallel array operations and stencils
  - Boost uBlas: matrix library
  - ITL and IETL: iterative algorithms for linear systems
  - Boost Graph library: graph algorithms
  - POOMA: particle and field simulations
  - ALPS: simulations of classical and quantum lattice models
- see http://oonumerics.org/ and http://www.boost.org/ for an overview

Blitz++

- was developed by Todd Veldhuizen
- available from http://oonumerics.org/blitz/
- was the first application of expression templates

- contains optimized classes for
  - d-dimensional arrays:

    ```
    template <class T, int D>  class Array<T,D>;
    ```

    is a d-dimensional array
  - short vectors of fixed length

    ```
    template <class T, int N> class TinyVector<T,N>;
    ```

    is a short vector of length N
  - small matrices, ...
- supports mathematical expressions with arrays

Blitz++: TinyVector

- TinyVector uses template meta programs to achieve optimal performance

- Example: a ray reflection: with $\vec{n} = (0,0,1)$

$$\vec{o} = \vec{i} - 2(\vec{i} \cdot \vec{n})\vec{n}$$

- can be coded in Blitz++ as:

  -
    ```
    TinyVector<double,3> i;
    i = …// set i
    TinyVector<double,3> n=0,0,1;
    TinyVector<double,3> o=i-2*sum(i*n)*n;
    ```
- This will give the *optimal* code:

  -
    ```
    o[0]=i[0];
    o[1]=i[1];
    o[2]=-i[2];
    ```

Blitz++: array expressions

- Consider the free energy and inner energy of a statistical system with energy levels energy[i]:

$$F = -T \log \sum_i \exp(-E_i / T)$$

$$U = \sum_i E_i \exp(-E_i / T) \Big/ \sum_i \exp(-E_i / T)$$

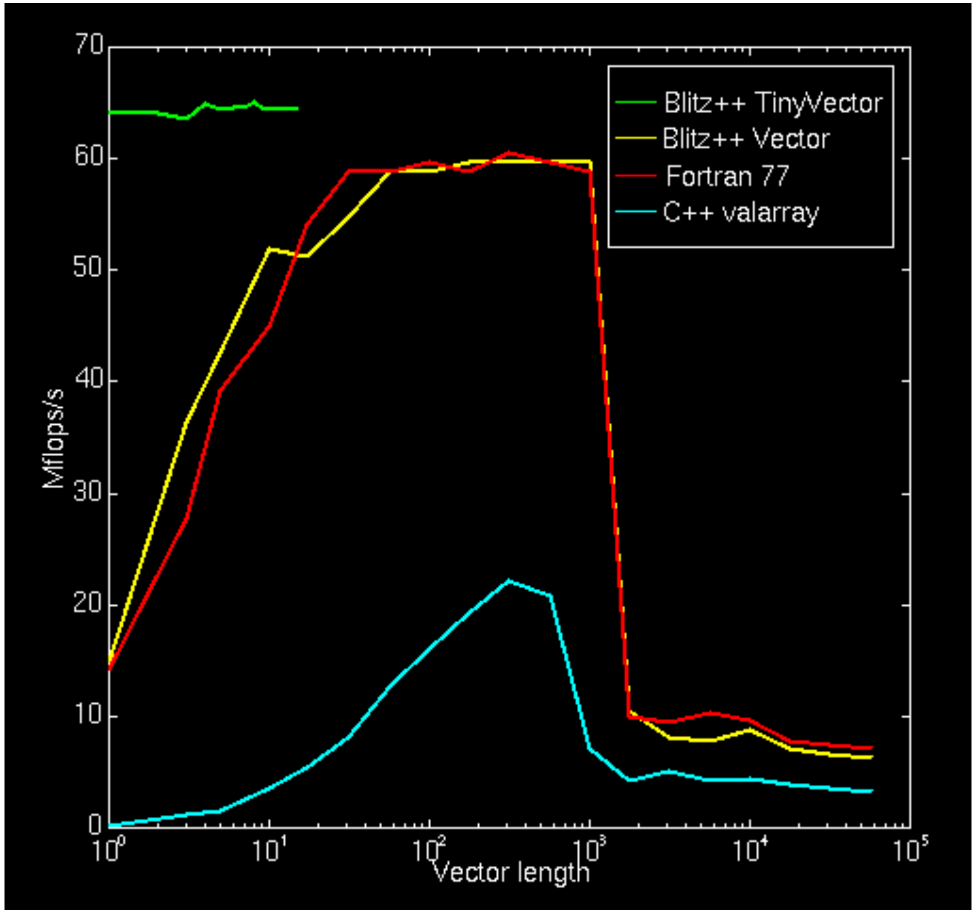
- Can be coded in Blitz++ as:

  -
    ```
    double F(const Array<double,1>& energy,double T) {
      return -T*log(sum(-exp(energy/T)));
    }
    ```

  -
    ```
    double U(const Array<double,1>& energy,double T)  {
      return sum(energy*exp(-energy/T)) /
             sum(exp(-energy/T));
    }
    ```
- That’s easy, isn’t it?

Blitz++: array benchmark

- Faster than simple-minded C++, about as fast as Fortran-77



Blitz++: Range objects

- A 1-D heat equation solver in would be:

  -
    ```
    Array<double,1> oldT(N), newT(N);
    for (int iter=0; iter<niter; ++iter) {
      for (int i=1;i<N-1;++i)
        newT(i)=oldT(i)+c * (oldT(i-1)+oldT(i+1)-2*oldT(i));
      swap(oldT,newT);
    }
    ```

- Using a `Range` object the inner loop is simplified to:

  -
    ```
    Array<double,1> oldT(N), newT(N);
    Range I(1,N-2);
    for (int iter=0; iter<niter; ++iter) {
       newT(I)=oldT(I)+c * (oldT(I-1)+oldT(I+1)-2*oldT(I));
       swap(oldT,newT);
    }
    ```

- The Range expression `newT(I)=` … gets expanded into an optimzied loop

Stencils

- The expression `oldT(I-1)+oldT(I+1)-2*oldT(I)` is quite common as a second order derivative
- Blitz++ allows “stencils” to be coded to simplify the use of such expressions
- Example: heat equation in 2D:

  ```
  Array<double,2> oldT(N,N), newT(N,N);
  Range I(1,N-2); // the interior region
   // create a view of the interior
  Array<float,2> oldI = oldT(I,I), newI = newT(I,I);
  for (int iter=0; iter<niter; ++iter) {
      // apply the stencil
      newI = oldI+c * Laplacian2D(oldI);
      swap(oldT,newT);
  }
  ```

- That’s easy now!

Blitz++: benchmark of 3D wave propagation

- http://www.oonumerics.org/blitz/benchmarks/acou3d.html

	Time [in seconds]	Lines of code
Blitz++	200	9
Fortran 90	294	12
Fortran 77	168	18
Blitz++ (tuned)	123	8
Fortran 90 (tuned)	140	21
Fortran 77 (tuned)	120	27

POOMA: particles and fields

- was originally developed by the Advanced Computing Lab of the Los Alamos National Laboratories
- has classes for parallel simulations of fields and particles
- contains expression templates and stencils like Blitz++
- can be used for
  - partial differential equations (e.g. parallelized heat equation)
  - Particle-in-Cell codes for particle simulations
- available from http://www.codesourcery.com/pooma/pooma/

Parallel diffusion equation using POOMA II

```
Interval<2> I2(N,N); // 2D index domain NxN
Loc<2> blocks(2,2);  // domain decomposition into 2x2 blocks
UniformGridPartition<2> partition(blocks);
UniformGridLayout<2> layout(I2, UniformGridPartition<2>(blocks),
    DistributedTag());
//Create containers to store temperature
Array<2,double, MultiPatch<UniformTag,Remote<Brick> > >
  oldT(layout), newT(layout);

Interval<1> I(1, N-2);
Interval<1> J(1, N-2);

// initialize variables
…
// iterate
for (i=0;i<iter;++i) {
  newT(I,J) = oldT(I,J) + c*0.25*(oldT(I+1,J) + oldT(I-1,J) +
                                  oldT(I,J+1) + oldT(I,J-1));
  oldT = newT;
}
```

The Boost uBlas library

- available from http://www.boost.org/
- contains
  - 1-d vectors
  - 2-d matrices:
    - Fortran-style, C-style, arbitrary style layout
    - Sparse and dense matrix types
  - full BLAS functionality
  - Simple interface to BLAS and LAPACK under development
- Everything is in the namespace `boost::numeric::ublas`

Boost uBlas matrices

- Boost uBlas is very flexible and has classes for
  - Special matrices
    - Zero matrix
    - Identity matrix
  - Dense matrices
    - General
    - Symmetric
    - Hermitian
  - Sparse matrices
    - Banded
    - A number of sparse storage formats

- In addition you can choose, if you wish, the
  - Storage layout (Fortran or C-style)
  - Storage container

The Boost uBlas matrix class

- Is a class for dense matrices of arbitrary type T:

  -
    ```
    template <class T, class F=row_major>
    class matrix;
    ```

- The layout can optionally be changed from `row_major` to `column_major` (Fortran style) with the second argument

- A few useful functions:
  - `matrix<double> a(3,3);` // constructor for a 3x3 matrix
  - `a(1,2)=4.;` // sets matrix element $a_{23} = 4$
  - `std::cout << a;` // prints the matrix
  - `double* p = a.data();` // gets a pointer to the data for use in a BLAS call
  - `c=mult(a,b);` // calculates c=a*b

- Look at the documentation for more functions and examples

LAPACK wrappers

- The Boost uBlas contains dense matrices which have the Fortran data layout
- The uBlas wrapper project in contains easier wrapper functions to some BLAS and LAPACK functions.
  - Your contributions to this open source project are appreciated

- uBlas wrapper:

```
template <class MatrixA, class MatrixB>
int gesv(MatrixA& a, MatrixB& b);
```

- Fortran LAPACK:

  -
    ```
    void dgesv_(const int& n, const int& nrhs, double da[],  const
     int& lda, int ipivot[], double db[], const int& ldb, int& info);
    ```

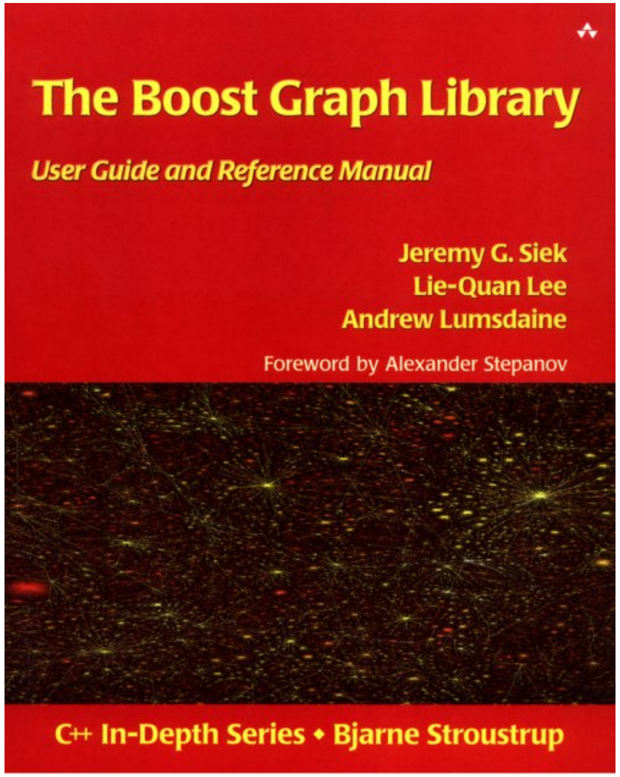
  -
    ```
    void sgesv_(const int& n, const int& nrhs, float da[],  const int&
     lda, int ipivot[], float db[], const int& ldb, int& info);
    ```
  - …

Graph problems: The Boost Graph Library (BGL)

- The BGL is a generic library for graph problems, modeled after the standard template library

- Is a part of Boost, available from http://boost.org/

- A very nice manual exists as a book


The Boost Graph Library
User Guide and Reference Manual
Jeremy G. Siek
Lie-Quan Lee
Andrew Lumsdaine
Foreword by Alexander Stepanov
C++ In-Depth Series • Bjarne Stroustrup

ALPS: Monte Carlo simulations

- is a library for the automatic parallelization of Monte Carlo simulations
- performs
  - parameter input
  - checkpoints and result files in hardware independent format
  - parallelization of MC simulations
  - load balancing
  - collection and evaluation of results
  - reliable error estimates
- develop and debug your code on a workstation
- link it on a parallel machine and it will run in parallel
- Available from http://alps.comp-phys.org/

Libraries you should install for your future work

- BLAS or ATLAS from http://netlib.org/
- LAPACK from http://netlib.org/
- Boost from http://boost.org/
- Blitz++
  - Older releases from http://oonumerics.org/blitz/
  - New releases from http://sourceforge.net/project/showfiles.php?group_id=63961
- FFTW from http://www.fftw.org/

- If you need a specific library, look at
  - Numerical libraries in Fortran or C: Netlib http://netlib.org/
  - Numerical libraries in C++: http://oonumerics.org/
  - General C++: Boost http://boost.org/