Generic Software Components for Scientific Computing
(or, How to Have Your Abstractions and Your Performance Too)

Andrew Lumsdaine

Laboratory for Scientific Computing
University of Notre Dame

http://lsc.nd.edu/research/mtl/
Overview

- Introduction
- Software: The good, the bad, the ugly
- Generic programming
- The Matrix Template Library
- The Basic Linear Algebra Instruction Set / Fixed Algorithm Size Template Libraries
- Performance results
- Conclusions
What This Talk is About

- Performance and abstraction are not mutually exclusive
- A new way of thinking about and building software systems
- Some real libraries built in this new way
What This Talk is NOT About

- Yet Another Object Oriented Math Library (YAOOML)
- Silver bullets
- One true programming language
- My language is better than your language
Software

- Applications in computational science are becoming software systems
- Not just “codes”
- Software engineering required
- Computer science needed
Software

- Writing good software is hard
- Writing bad software is easy
- Product is ephemeral (what is \textit{good} software, anyway?)
- Even with abstractions there are more than a brainful of (i.e., seven things) to worry about
- Many system issues are relegated to be “fixed in the software”
Software Engineering

- Is it engineering at all?
- Typically (ideally?)
  - Understand the problem to be solved
  - Propose and analyze a solution
  - Design a software system to realize the solution
  - Implement, debug, test the software system
  - Deliver, maintain, live happily ever after
The Famous (Infamous?) Waterfall Model

- Requirements
- Design
- Implementation
- Testing
- Maintenance
Managing Software Complexity

- Abstraction is necessary to manage complexity
- The brain fills up at seven plus or minus two
- Example – numerical linear algebra
  - matrix-vector
  - array
  - number
Abstraction

- Interface specification between layers or modules
- One layer is not allowed to mess with the internals of another layer

Examples
- Procedural
- Structures
- Objects
Abstraction Goals for the Computational Scientist

- Separate interface from implementation (computational recipe)
- Results
  - Improved modularity
  - Better maintainability
  - Better reliability
  - Better productivity
  - Better performance
Performance Goals for the Computational Scientist

- Peak (hardware advertised)
- Achievable
- Conventional wisdom (changing)
  - Fortran required
  - Vendors must supply BLAS
  - Assembly language
Good Abstractions do not Mean Poor Performance

- Abstraction is clearly valuable for developing quality software systems
- The question is how to use abstractions in such a way as not to hinder (or better yet to actually enable) high levels of performance
- (Hint: Flatten the abstractions for machine consumption)
Claims

- Computer does not execute C or Fortran or C++ or Ada
- Source language is largely irrelevant from the performance point of view
Some Proposed Solutions

- Better compilers
- Lazy evaluation
- Templates (in the “Templates Book” sense)
- Program with abstractions
But I Already Program with Abstractions!

- Software abstractions tend to lose their abstractness once a concrete type is defined.
- Let us use the term **Concept** to refer to the abstraction in the abstract.
- Recurring theme in software development – write once algorithms.
- Abstract base class in C++ was an attempt to provide programming with concepts (not successful).
Abstraction? Concepts?

● Webster says:
  – Considered apart from concrete existence
  – Not applied or practical
  – Difficult to understand
  – Impersonal, as in attitude or views
  – To consider (a quality, for example) without reference to a particular example or object
Programming with Concepts = Generic Programming

- Programming with concepts leads naturally to the notion of generic components
Why Generic Components

- Humans are good at building some very complex things very well.
- Consider an automobile or a jetliner.
- How? Where does the “conceptual integrity” reside?
- No one designs the jet or even any one subsystem completely.
Human organizations have one individual at the top, but

- This person knows the least about the technical details of the end product (Dilbert principle)
The person at the top manages the process from which the product emerges

- Software engineering is analogous to industrial engineering in this sense
Co-Problem of Complex Systems

- Designing the process and the organization to produce the end product
- The end result is an emergent product of the co-problem
- Humans seem to be good at designing processes
- But how process gives rise to product is not well understood
Human Intuition for Software Processes is not on Par with Hardware

- Humans have been building hardware much longer
- Hardware is more quantifiable and measurable
- Ambiguity of language (Tower of Babel ad infinitum)
- One key idea: interchangeable components
Software Engineering

- Product must emerge from **process**
- Emergence of product from process must be better understood
- Co-problem needs to be properly formulated
- Local rules $\rightarrow$ global behavior
- New mathematics for large systems?
What Should Good (Scientific) Software Look Like Then?

- Abstractions
- Interchangeable components
- High performance
Generic Programming

- Not about language features
- Creation of a systematic organization of abstract and efficient software components
Generic Programming

- Identify useful and efficient algorithms
- Find their generic representation
- Derive a set of (minimal) requirements that allow these algorithms to run and run efficiently
- Construct a framework based on classifications of requirements
The Standard Template Library is an important example of generic programming.

template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init)
{
    while (first != last)
        init = init + *first++;
    return init;
}
Generic Programming is Programming with Concepts

- A concept is a minimal set of operations on a family of types which are required for an algorithm to work

```cpp
template <class InputIterator, class T>
T accumulate(InputIterator first,
             InputIterator last, T init)
{
    while (first != last)
        init = init + *first++;
    return init;
}
```
Generic Programming is Programming with Concepts

- In this example `accumulate()` can be used with any data type satisfying the `concept` of InputIterator
- An InputIterator can be de-referenced
- An InputIterator can be incremented
- This is the minimal set of requirements
Generic Programming is Programming with Concepts

- `accumulate()` can be used with any data type satisfying the *concept* of InputIterator

```cpp
double x[10];
vector<double> y(10);
list<complex<double>> z(10);

a = accumulate(x, x+10, 0.0);
b = accumulate(y.begin(), y.end(), 0.0);
c = accumulate(z.begin(), z.end(), 0.0);
```
Examples of Generic Programming for Scientific Computing

- The Matrix Template Library (MTL): Generic Programming for Numerical Linear Algebra
- The Basic Linear Algebra Instruction Set and the Fixed Algorithm Size Template (BLAIS and FAST) Libraries: Generic Programming for High Performance
The Matrix Template Library: Generic Programming for Numerical Linear Algebra

- Identify useful and efficient algorithms
- Find their generic representation
- Derive a set of (minimal) requirements that allow these algorithms to run and run efficiently
- Construct a framework based on classifications of requirements
Useful and Efficient Algorithms for Numerical Linear Algebra

- In the abstract sense
- Guided by the mathematical definition of linear algebra
A complete set of linear algebra functionality

<table>
<thead>
<tr>
<th>Generic algorithm</th>
<th>Abstract operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale()</td>
<td>$\alpha \cdot x \in V$</td>
</tr>
<tr>
<td>add()</td>
<td>$(\alpha \cdot x + \beta \cdot y) \in V$</td>
</tr>
<tr>
<td>mult()</td>
<td>$A : V \rightarrow V$</td>
</tr>
<tr>
<td>norm()</td>
<td>$|x|$</td>
</tr>
<tr>
<td>dot()</td>
<td>$\langle x, y \rangle$</td>
</tr>
<tr>
<td>transpose()</td>
<td>$A^* : V^* \rightarrow V^*$</td>
</tr>
</tbody>
</table>
Conjugate Gradient Algorithm

for $i = 1, 2, \ldots$

solve $Mz^{(i-1)} = r^{(i-1)}$

$\rho_{i-1} = r^{(i-1)T}z^{(i-1)}$

if $i = 1$

$p^{(1)} = z^{(0)}$

else

$\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$

$p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)}$

$q^{(i)} = Ap^{(i)}$, $\alpha_i = \rho_{i-1}/p^{(i)T}q^{(i)}$

$x^{(i)} = x^{(i-1)} + \alpha_ip^{(i)}$, $r^{(i)} = r^{(i-1)} - \alpha_iq^{(i)}$

check convergence

end
while (! iter.finished(r)) {
    M.solve(r, z);
    rho = dot_conj(r, z);
    if (iter.first()) copy(z, p);
    else { beta = rho / rho_1;
        add(z, scaled(p, beta), p); }
    mult(A, p, q);
    alpha = rho / dot_conj(p, q);
    add(x, scaled(p, alpha), x);
    add(r, scaled(q, -alpha), r);
    rho_1 = rho;
    ++iter;
}
template <class Matrix, class VecX, class VecY>
void mult (Matrix A, VecX x, VecY y) {
    typename Matrix::iterator i;
    typename Matrix::OneD::iterator j;
    for (i = A.begin(); i != A.end(); ++i)
        for (j = i->begin(); j != i->end(); ++j)
            y[j.row()] += *j * x[j.column()];
}
Containers for Linear Algebra

- Concrete representations of members of \( V \) (vectors) and linear transformations (matrices)

- Vectors are relatively straightforward (one dimensional container of type \( F \), similar to STL \texttt{vector})

- Matrices are more involved because two-dimensional object must be implemented with a one-dimensional memory space
Requirements for Numerical Linear Algebra Algorithms

- As with the STL, use *iterators* to traverse through a container.
- A matrix representation can be abstractly thought of as a *container of containers*.
- Use *iterators* and *2-dimensional iterators* to traverse the matrix.
- Almost all matrix types in use today can be implemented with this interface.
Containers for Linear Algebra:

Properties

Storage: e.g., dense contiguous, compressed sparse

Basic (elemental) type: e.g., float, complex<double>

Orientation: e.g., row, column, diagonal

Shape: e.g., symmetric, triangle, banded
MTL Containers for Linear Algebra

- MTL matrices are built by template composition of
  - Basic numeric type (precision) (x6)
  - One-dimensional container (x5)
  - Two-dimensional container (x2)
  - Orientation (x3)
  - Shape and packing (x8)
- Approx 1440 matrix types implemented in 16 classes
The MTL Framework

- Algorithms
- Iterators
- Containers
- Function objects
- Adaptors
Example Use: Block LU Algorithm

- Perform point-wise LU to get \( L_{11}, U_{11}, \) and \( L_{21} \).
- Do a triangular solve to get \( A_{12} \).
- Do matrix product \( A_{22} \leftarrow L_{21} \times A_{12} \).
The Block LU Implementation

```c
void block_lu(Matrix& A, Pvector& ipvt) {
    Pvector pivots(BF);
    for (int j = 0; j < min(M, N); j += BF) {
        int jb = min(min(M, N) - j, BF);
        // set up the submatrices A0, A1, A2
        //     L11, A12, A21, A22 ...
        lu_factorize(A1, pivots);
        multi_swap(A0, pivots);
        if (j + jb < M) {
            multi_swap(A2, pivots);
            tri_solve(L11, A12, left_side());
            if (j + jb < M)
                mult(scaled(A21,-1),A12,A22);
        }
    }
}```
The BLAIS and FAST Libraries: Generic Programming for High Performance

- Identify useful and efficient algorithms
- Find their generic representation
- Derive a set of (minimal) requirements that allow these algorithms to run and run efficiently
- Construct a framework based on classifications of requirements
High performance amidst the use of abstractions

Abstract Tuning: Parameterize well known optimization techniques

Static Polymorphism: Functions must be selected at compile time and inlined away

Lightweight Object Optimization: Structures must be ripped apart and en-registered

Follow Coding Guidelines: Enable the C++ compiler to make the above optimizations
Performance Digression

- It is easy to get high levels of performance if you define “high” in a suitable (but vague) fashion
  - “Equivalent to C / Fortran”
  - “Equivalent to hand-written”
  - “Within a factor of 2 of C / Fortran”
- Vendor-tuned are typically (but not always) the best
- MTL performance goal is to offer best possible performance
Peak Performance

- Tuning process is fairly well understood
- Make use of architecture features that are provided for high performance
  - Cache (improve temporal and spatial locality)
  - Instruction pipelining
- These processes can be parameterized
- Create abstractions to handle the optimizations
Useful Algorithms for High Performance

Important optimizations to perform in the C/C++ code:

- Loop unrolling
- Register-level blocking

The blocking sizes are machine dependent, so the optimization scheme must be flexible and parameterized, which is impossible to do in C or Fortran.
Generic Representation of High Performance Algorithms

It is impossible to express variable degrees of unrolling and blocking in C and Fortran

// unroll by two
y[0] += a * x[0];
y[1] += a * x[1];

// unroll by three
y[0] += a * x[0];
y[1] += a * x[1];
y[2] += a * x[2];
Previous Solutions: PHiPAC and ATLAS

- Search scripts find best blocking factors
- Code generation system customizes the code
- Result is portable high performance
- Complex software system
- Hard to maintain and/or modify (the numerical code is controlled indirectly)
Generic Representation of High Performance Algorithms

- With *template meta-programming* techniques, **variable** degrees of unrolling can be directly expressed
- Made possible by integer template parameters

```cpp
template <class T, int M>
class X {
  ...
};
```
The Fixed Algorithm Size Template (FAST) Library

- Essentially STL for fixed (at compile time) size computations
- A combination of generic programming with template meta-programs
- Suitable for small sized, performance critical kernels
- Demonstrates that extra abstraction levels do not hinder performance
Comparison of STL and FAST

// STL
int len = 4;
int* x = new int(len); int* y = new int(len);
fill(x, x+len, 1); fill(y, y+len, 3);
std::transform(x, x+len, y, y, plus<int>());

// FAST
const int LEN = 4;
int* x = new int(LEN); int* y = new int(LEN);
fill(x, x+LEN, 1); fill(y, y+LEN, 3);
fast::transform(x, cnt<LEN>(), y, y, plus<int>());
Definition of fast::transform()

- Recursion is used instead of loops
- Recursion depth is fixed and each call becomes inlined

```cpp
template <int N, class InIter1, class InIter2, class OutIter, class BinOp>
OutIter
transform(InIter1 in1, cnt<N>(), InIter2 in2,
    OutIter out, BinOp binary_op) {
  *out = binary_op (*in1, *in2);
  return transform(++in1, cnt<N-1>(), ++in2, ++out, binary_op);
}
```
Basic Linear Algebra Instruction Set (BLAIS)

- Linear algebra kernels for fixed sized computations.
- Complete expansion results in no loops. Just as good as hand coded unrolling.
- Presents a simple and elegant interface.
- Simple implementation layered on the Fixed Algorithm Size Template (FAST) library.
- Template metaprograms can be elegant!
// General Case

template <int M, int N>
struct mult {
    template <class ColIter, class IterX, class IterY>
    mult(ColIter col_iter, IterX x, IterY y) {
        add<M>(scl((*col_iter).begin(),*x), y);
        mult<M, N-1>(++col_iter, ++x, y);
    }
};

// N = 0 Case
...
void mult (MatA& A, MatB& B, MatC& C) {
    while (A_k != A.end()) {
        while (B_j != B.end()) {
            MatC::Block Cblock = *C_kj;
            while (B_ji != (*B_j).end()) {
                blais::mult(*A_ki,*B_ji,Cblock);
                ++B_ji; ++A_ki;
            } // cleanup K left out
            ++B_j; ++C_kj;
        } // cleanup N left out
        ++A_k; ++C_k;
    } // cleanup M left out
}
Dense Matrix-Matrix Performance
(UltraSPARC 170E)
Dense Matrix-Matrix Performance
(UltraSPARC 30)
Dense Matrix-Matrix Performance

(RS6000 590)
Dense Matrix-Vector Performance

(UltraSPARC 170E)
Sparse Matrix-Vector Performance
(UltraSPARC 170E)
Related Work

- Iterative Template Library (see MTL web site)
- Extended precision numeric type
- LAPACK interface (?!)
- Interval arithmetic (release imminent)
- Generic Graph Component Library (release imminent)
Future Work

- Generic Parallel Communication Library
- Generic Grid Generation
- New BLAS implementation
- Etc.
- Development tools for generic programming
Conclusions: Generic Programming Has Many Benefits

- Interchangeable components
- Huge (multiplicative) functionality delivered with small (additive) effort
- Elegance
- Extreme levels of performance
Conclusions: Performance

- Performance is a solved problem and is not a valid argument for the use of primitive programming languages.
- Performance portability is a solved problem. Libraries do not need to be supplied by vendors.
- Performance portable, functionally comprehensive libraries can be developed with reasonable scale of effort.