

On the Reusability and Numeric Efficiency of C++ Packages in Scientific Computing

Ulisses Mello

and

Ildar Khabibrakhmanov

IBM T. J. Watson Research Center
Yorktown Heights, NY, USA

Introduction

- We develop prototype code for numerically intensive applications in the area of fluid flow in porous media and seismic ray-tracing
- Fundamental questions:
 - In addition to the high-level design, should we use C++ for numerically intensive applications?
 - How efficient are C++ containers for Linear Algebra (LA) operations?

OO Paradigm

- Main Advantages of Object-oriented programming in numerically intensive applications:
 - Patterns and reusable components
 - Generic programming

C++ Techniques for NIA

- C++ for numerically intensive applications:
 - Expression Templates
 - Static polymorphism
 - Generative programming
 - Template metaprogramming
 - Linear Algebra (LA) native implementations
- How efficient C++ OO Packages are?

BTL- Benchmark for Templated Libraries Project

- Benchmark methodology and Generic Programming:
 - <http://www.opencascade.org/upload/87/index.html>
 - Laurent Plagne
- Benchmarked operations
 - Vector dot product (DOT)
 - Vector update (AXPY)
 - Dense vector-matrix multiply (GEMV)
 - Dense matrix-matrix multiply (GEMUL)

OO Techniques and Performance Issues

- Operator overload and temporary creation
- Static Polymorphism
- Generic Programming
- Template Metaprogramming
- Composition Closure Objects
- Expression Templates
- Generative Programming

OO Techniques and Performance Issues

- Operator overload and temporary creation

$$x = M * v$$

$$t = M * v; x = t$$

- Wrapper may also create temporary objs:

```
Vector operator * (Matrix& M, Vector& V) {  
    call BLAS_GEMV;  
}
```

Static Polymorphism

Dynamic Polymorphism (run time binding):

```
class shape { public: virtual void draw() const=0; }  
class circle : public shape { public: virtual void draw() const; }  
class square : public shape { public: virtual void draw() const; }  
std::vector<shape*> shapeCollection;
```

- **Problems: indirection, no optimization, no inlining, pipeline branch**

Static Polymorphism (compile time binding):

```
template <typename shape> void draw (shape const& o) { o.draw(); }  
class circle { public: void draw() const; }  
class square { public: void draw() const; }  
circle c; square s;  
draw<circle>(c);  
draw<square>(s);
```

- **Problems: homogenous collection**

Generic Programming

```
Template <class Type, class Container>
Type sum(const Container& c) {
    Container::const_iterator it0=c.begin();
    Container::const_iterator it1=c.end();
    Type s = 0;
    for(;it!=it1;++it) s += *it;
    return s;
}

// example
vector<int> vi;
list<double> ld;

int si = sum<int,vector<int> >(vi);
double sd = sum<double, vector<double> >(ld);
```

**Templates for the Solution of
Linear Systems: Building Blocks
for Iterative Methods (Miscellaneous
Titles in Applied Mathematics Series
No 43)**

by [Richard Barrett](#) (Editor),
[Henk Van Der Vorst](#) (Editor),
[Roldan Pozo](#) (Editor),
[Jack Dongarra](#), [Victor Eijkhout](#),
[Charles Romine](#), [Henk van der Vorst](#)

Template Metaprogramming

```
template<int N>
struct Factorial {
    enum { value = N * Factorial<N-1>::value };
};

// template specialization
template<> struct Factorial<1> { enum { value = 1 }; };

// Example use const int fact5 = Factorial<5>::value;
```

- 120 at compile time
- Very slow compilation time ...

Template Metaprogramming and Loop Unrolling

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

At compile time this code expands to:

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

Composition Closure & ET

- Composition Closure Objects

$$X.operator=(MVmul(M, V))$$

- Expression Templates

Simple Serial AXPY Example

- Using Standard Template Library - STL

```
inline void axpy(real coef, const stl::vector<real> & X,  
    stl::vector<real> & Y) {  
    for (int i=0;i<X.size();i++)  
        Y[i]+=coef*X[i];  
}
```

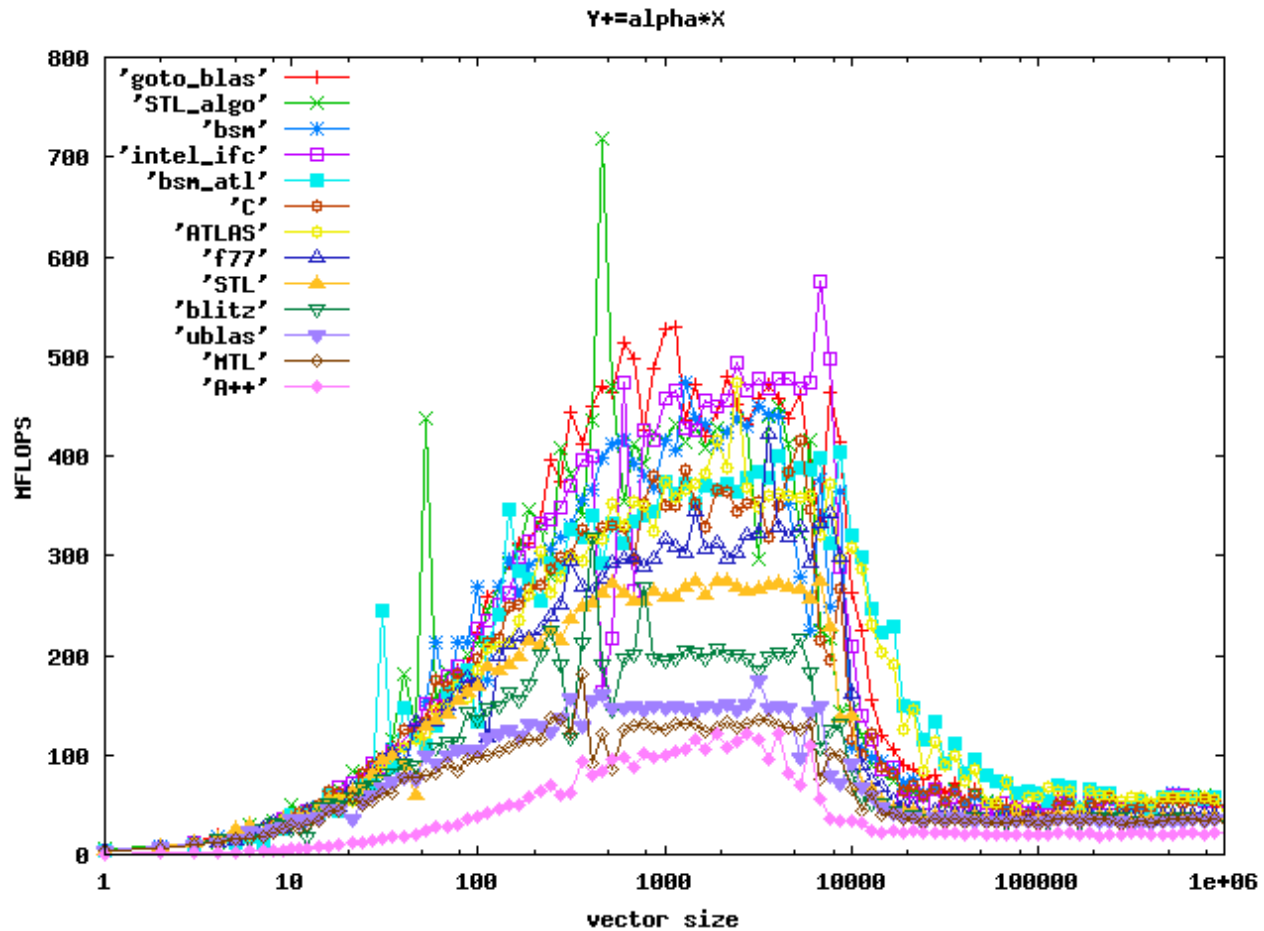
Results

- C++ compilers:
 - GNU g++ 3.2
 - Intel icc 7.0
- 8-node IBM Linux Cluster 1300
 - Red Hat 7.3 (non optimized kernel)
 - Marinet switch
 - PIII with 700MHz, 256KB of L2 (serial)
 - PIII with 1GHz, 512KB of L2 (parallel)

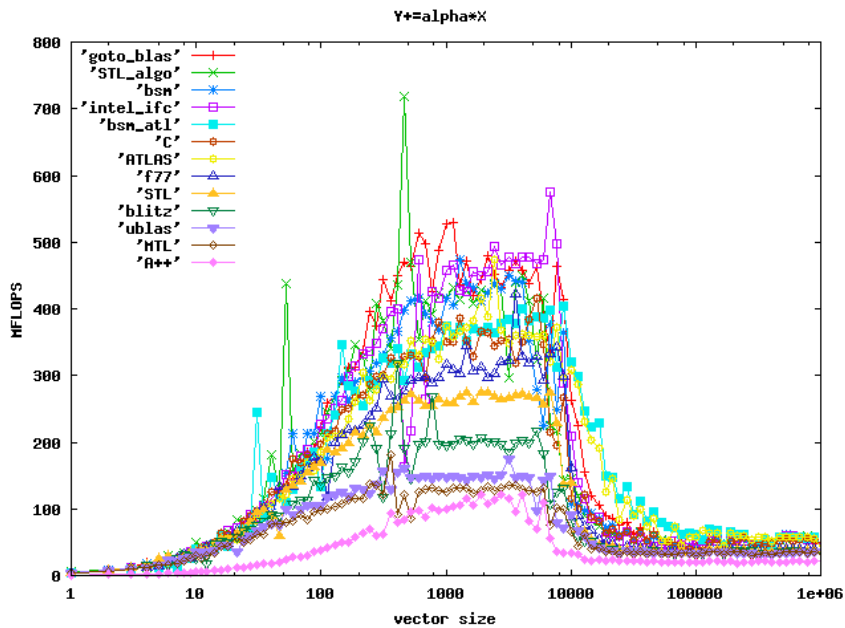
Serial OO Libraries

- A++
- Blitz++ (MP,ET)
- MTL
- Ublas (MP,ET)
- STL (vector<double> and valarray)
- Non-OO: Netlib's BLAS, goto blas, Atlas, raw C, raw fortran 77

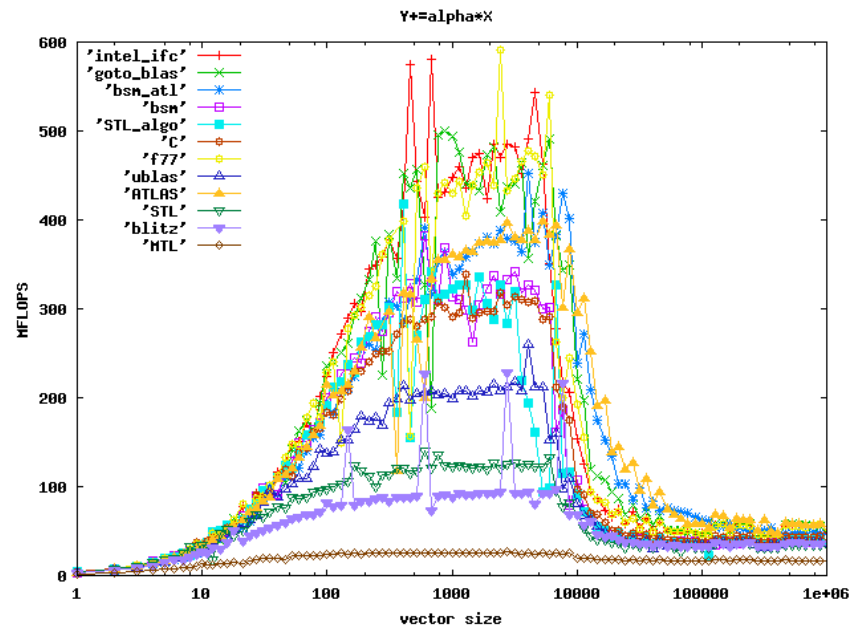
Vector Update - GNU



Vector Update (AXPY)

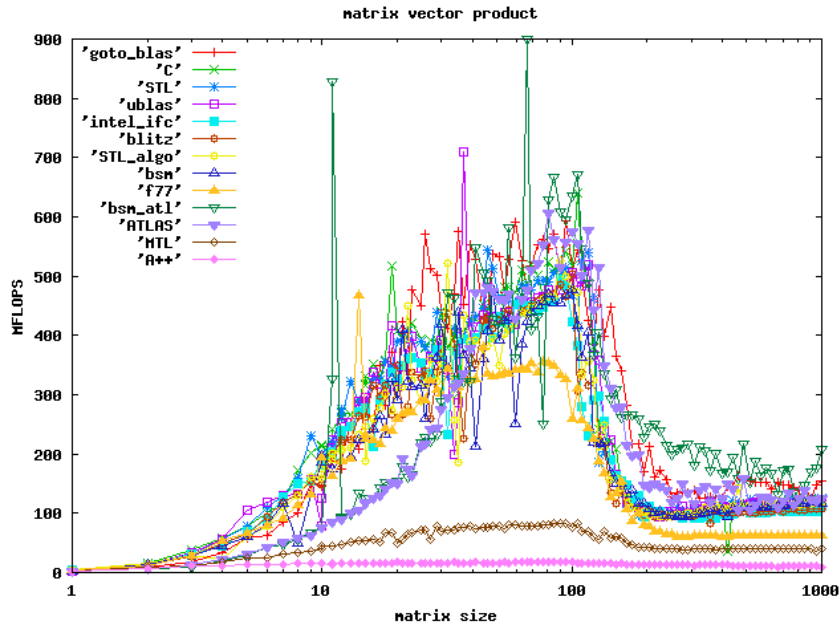


GNU

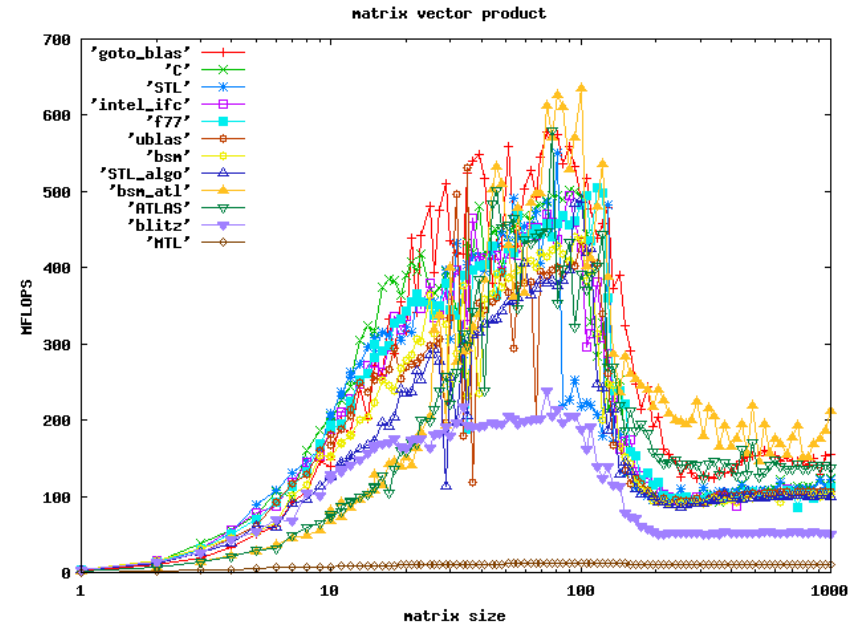


INTEL

Matrix-Vector Multiply (GEMV)

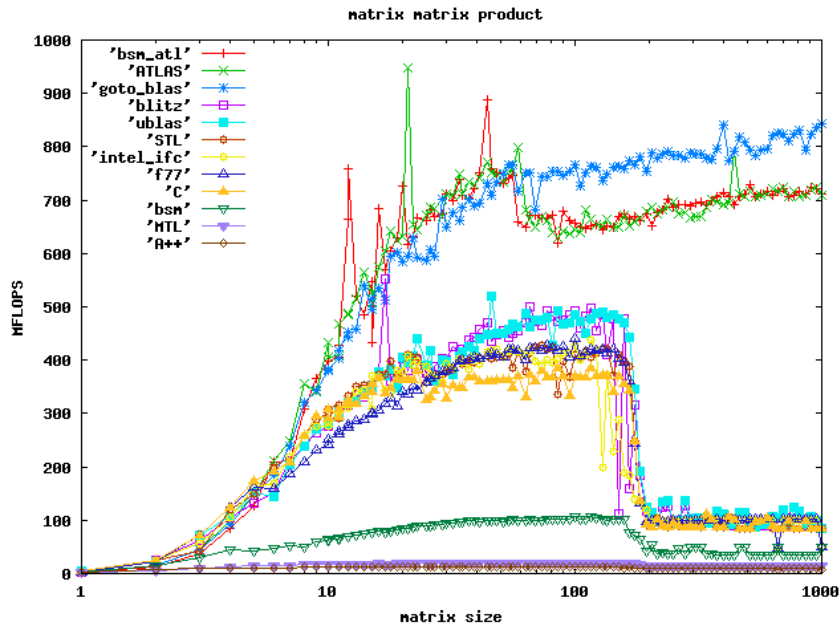


GNU

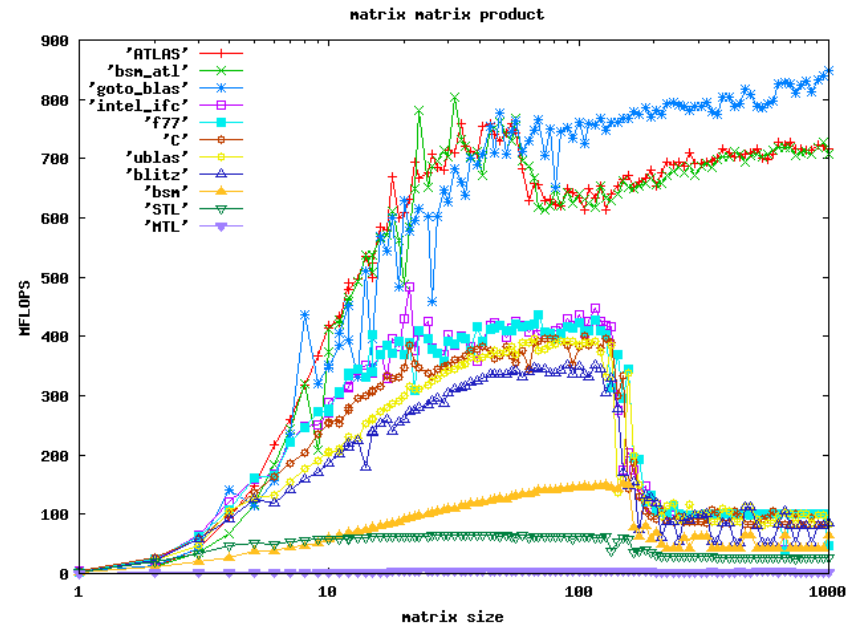


INTEL

Matrix-Matrix Multiply



GNU



Intel

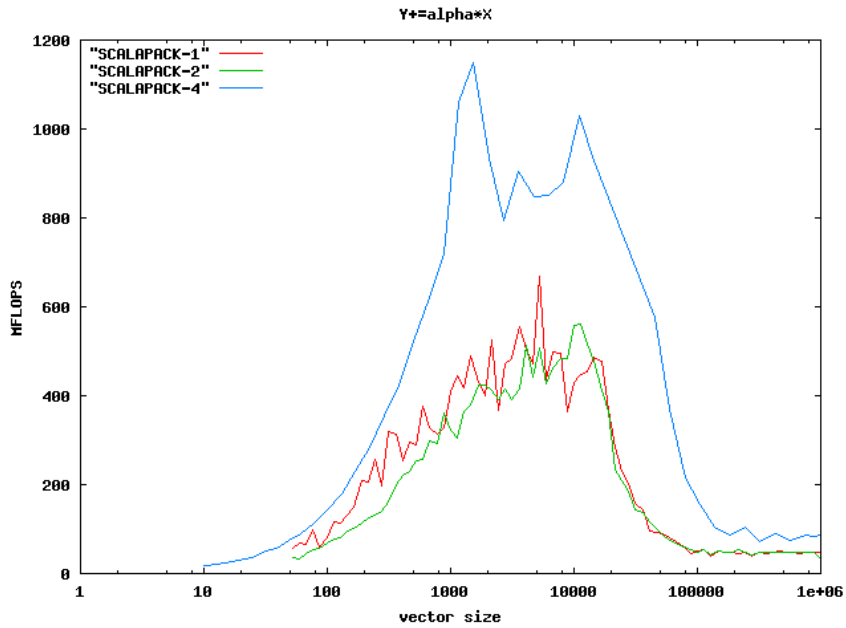
Parallel Libraries

- PLAPACK
- PETSc
- ScaLAPACK – non-OO

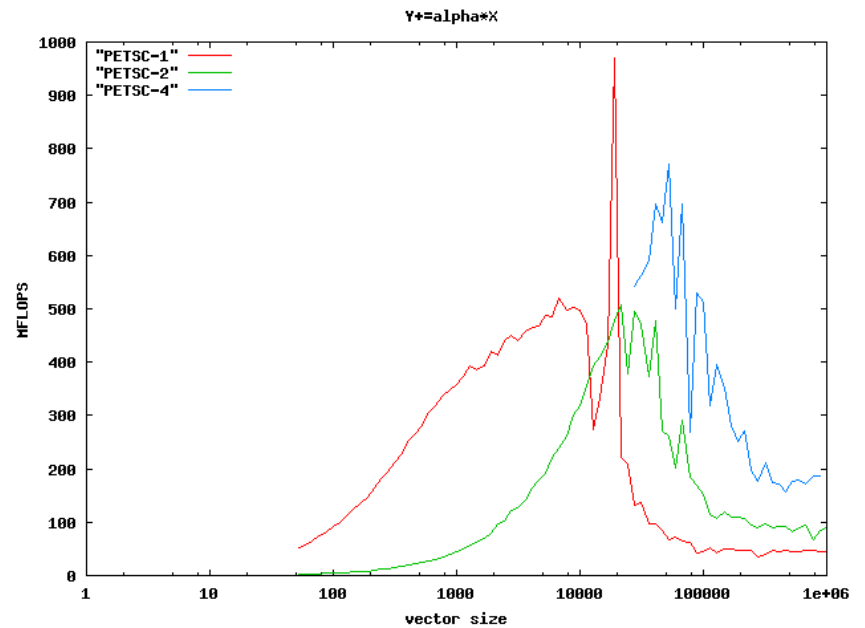
Not designed for LA operations:

- P++
- POOMA

Vector Update (AXPY)

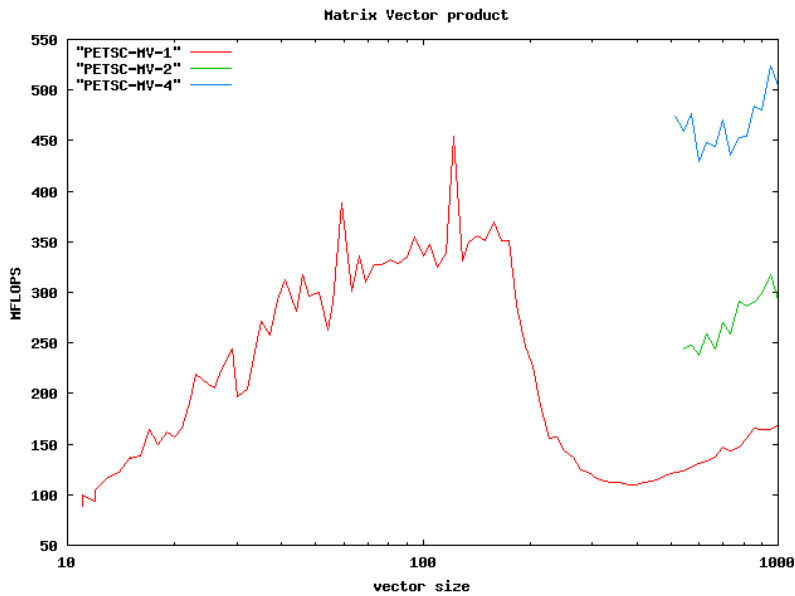


ScaLAPACK

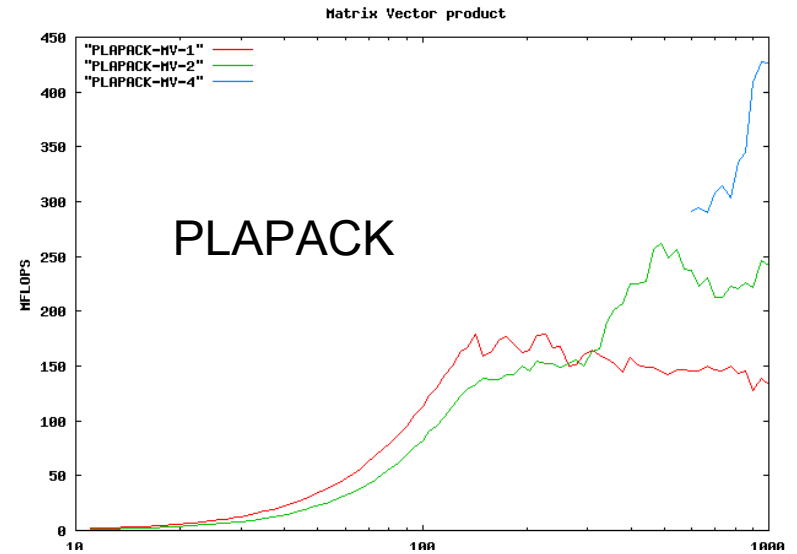


PETSc

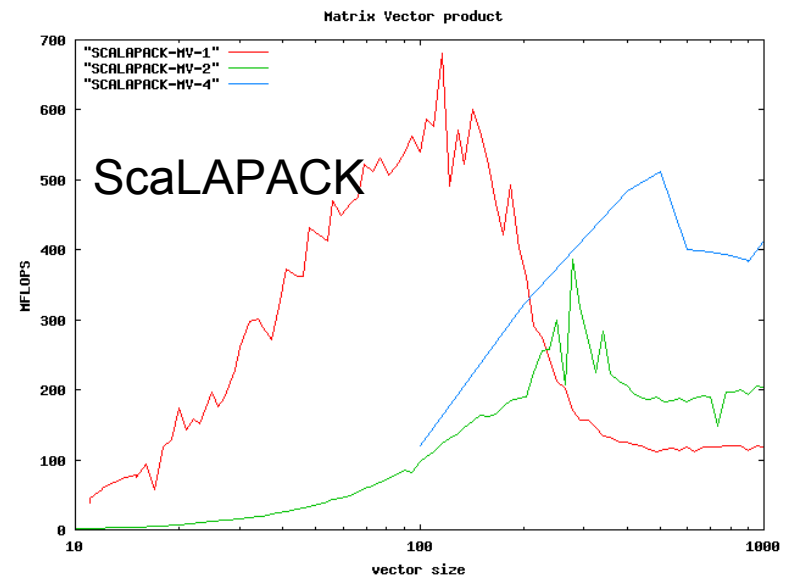
Matrix-Vector Multiply (GEMV)



PETSc

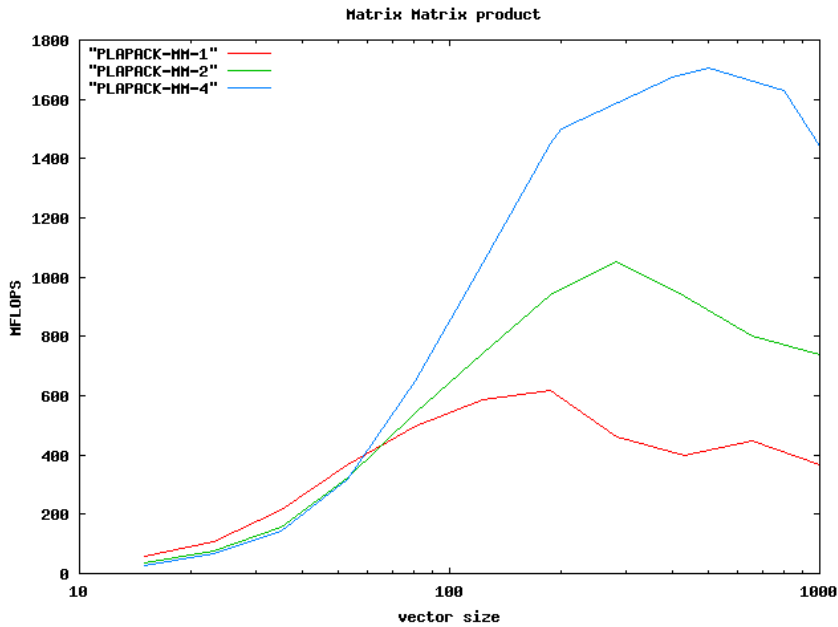


PLAPACK

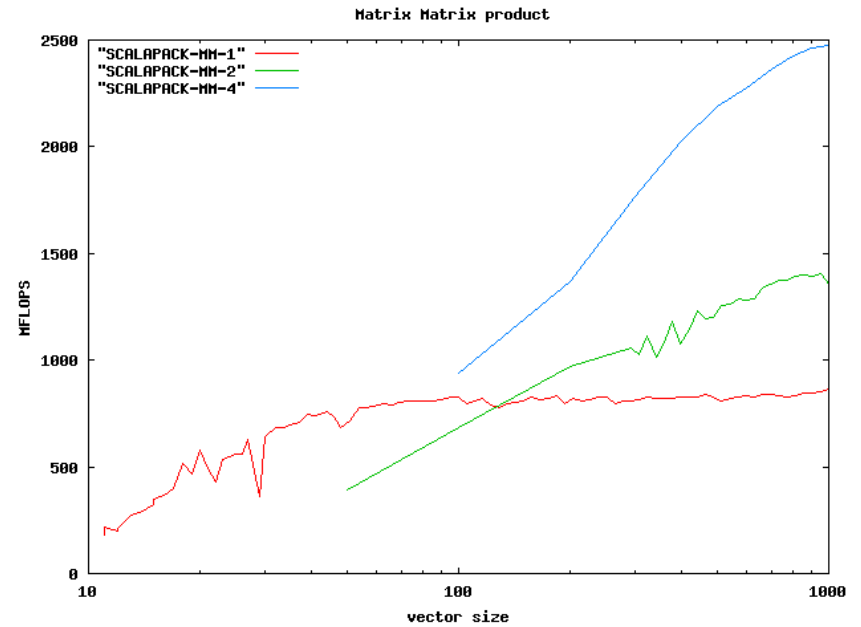


ScaLAPACK

Matrix-Matrix Multiply

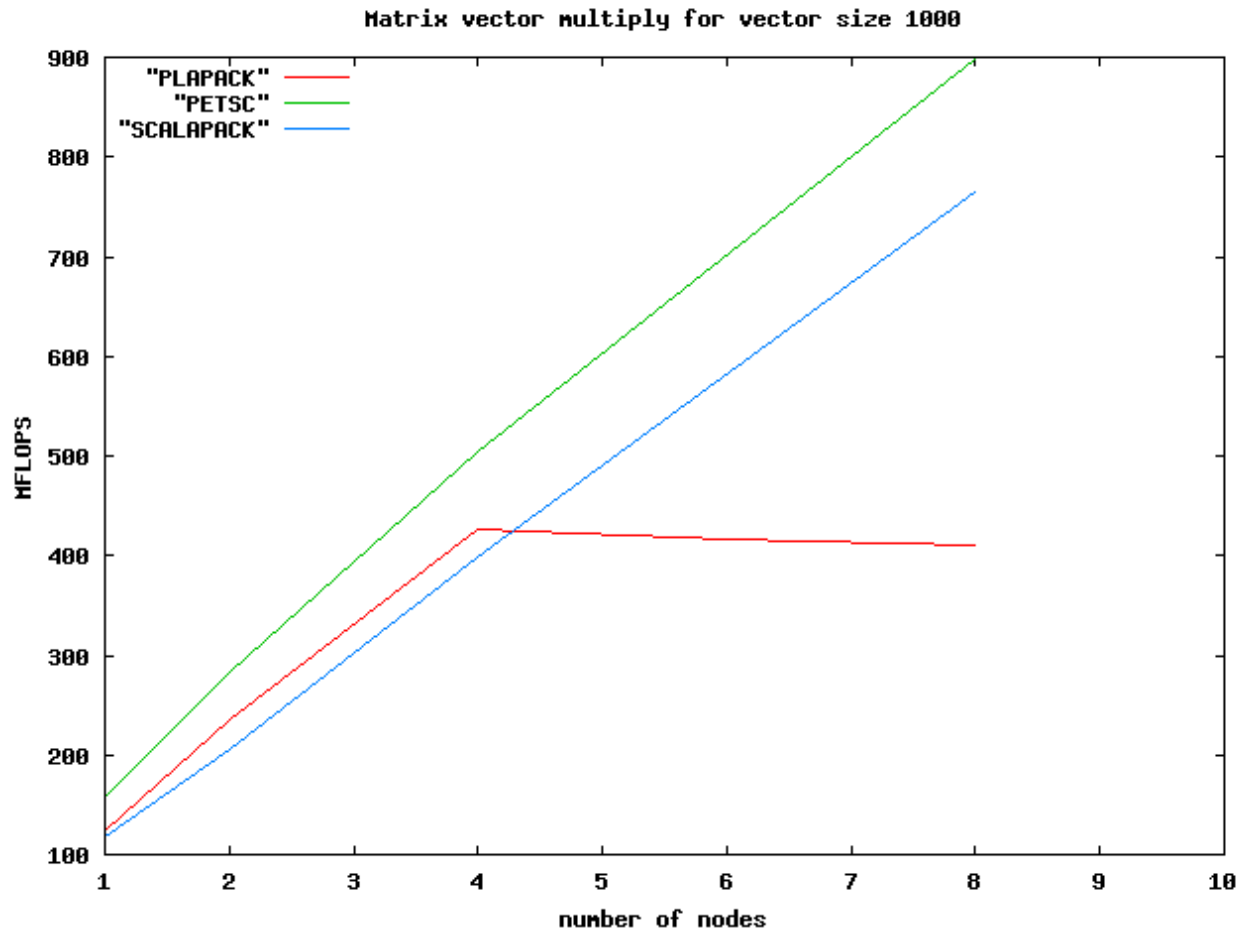


PETSc



ScaLAPACK

Matrix-Vector Multiply Scalability



Conclusions

- STL performance was quite good
- PETSc is the best OO package overall
- POOMA and A++/P++ are not designed to handle LA operations

- Wrap it ! GOTO uses assembly code
- Metaprogramming restricts reusability
- Strong compiler dependency