

C++ Expression Templates in an Embedded, Parallel, Real-Time Signal Processing Library

Edward M. Rutledge
MIT Lincoln Laboratory

Abstract

1.0 INTRODUCTION

In order to facilitate a smooth transition from an algorithm's linear algebra specification to its software implementation, a high performance signal processing library should provide software constructs that allow linear algebra to be easily translated into high performance code. Currently, C is the prevalent language for high performance signal processing libraries, largely because of concerns about the efficiency of C++. If it were not for these concerns, C++ would be a better choice, partly because of its expressiveness. Using C++, we can define vector and matrix classes and overloaded operators that allow linear algebra to be more easily and intuitively translated into code. However, C++ operator overloading is the source of much of the concern about the efficiency of C++. Traditional techniques of operator overloading incur added overhead that may be unacceptable in an embedded real-time system. C++ "expression templates," which are currently gaining popularity in the scientific computing arena, provide a solution to this problem.

In this presentation, we explore the impact of C++ expression templates on the simplicity and performance of an experimental, embedded, parallel, real-time signal processing library.

We demonstrate that expression templates can be as beneficial in the rapid development of embedded, parallel, real-time signal processing applications as they have proven to be in the rapid development of high performance scientific simulations. Our examples and experiments focus on architectures, problem sizes, and kernels relevant to radar and other array-sensor processing applications.

2.0 OVERVIEW OF C++ EXPRESSION TEMPLATES AND PETE

C++ overloaded operators typically return temporary objects containing the results of the operation. This technique incurs the overhead of creating and destroying the temporary objects, which include additional memory use, copy overhead, and possible overhead of allocating and freeing dynamic memory in the objects. Alternatively, C++ expression templates can be used to transform an arbitrary expression of objects such as vectors or matrices into a parse tree representation of that expression. This is done at compile time, and the parse tree object itself takes a small amount of memory, so little overhead is incurred at run time. With an expression's entire parse tree represented in a single object, the expression can be evaluated as a whole instead of being evaluated one operation at a time. This eliminates the need to produce temporary objects to hold intermediate results, creating a profound positive impact on the suitability of C++ operator overloading for embedded real-time systems, where efficient use of system resources is essential.

The Portable Expression Template Engine (PETE) is a free software package, developed at Los Alamos National Laboratory's Advanced Computing Laboratory, that allows programmers to add expression template capability to their C++ programs. PETE furnishes a utility for defining C++ operators that transform expressions of class objects into parse trees, and PETE furnishes functions that help in navigating these expression parse trees and help in efficiently evaluating expressions of element-wise operations. We use PETE to add expression template capability to the experimental C++ signal processing library referred to in this presentation.

3.0 IMPACT OF PETE ON CODE SIMPLICITY AND PERFORMANCE

Through the following examples and experiments, we show that PETE and C++ can combine to form an efficient and easy to use signal processing programming capability.

3.1 Simplicity

To show the impact of C++ and operator overloading on the simplicity and readability of signal processing code, we compare implementations of linear algebra expressions using our experimental C++ signal processing library with operator overloading to implementations using a C signal processing library.

3.2 Performance

To show the performance impact of PETE in our experimental signal processing library, we measure and compare the execution time of a linear algebra expression implemented in three different ways:

- using an optimized native C library,
- using our experimental C++ library with PETE operators and vector and matrix classes that evaluate the expression using an optimized native C library, and
- using our experimental C++ library with PETE operators and vector and matrix classes that evaluate the expression using PETE methods of element-wise evaluation.

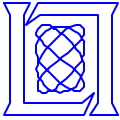
The linear algebra expression and problem size are extracted from an existing radar signal processing application.

The first two approaches are compared to show the overhead introduced by the C++/PETE approach. The second two approaches are compared to show the possible performance advantage of PETE element-wise expression evaluation over optimized native C library call expression evaluation, especially for evaluating expressions where many element-wise operations are chained together.

The experiments are run on both serial and distributed memory message passing architectures. In the latter case, we examine both data mappings that require inter-processor communication and those that do not.

4.0 FUTURE OPTIMIZATIONS

We also discuss some possibilities for further optimization of mathematical expressions, including determining the optimal order for performing a series of chained operations in an expression with respect to operation count and with respect to communication overhead. These optimizations would not be possible using traditional C++ operator overloading.



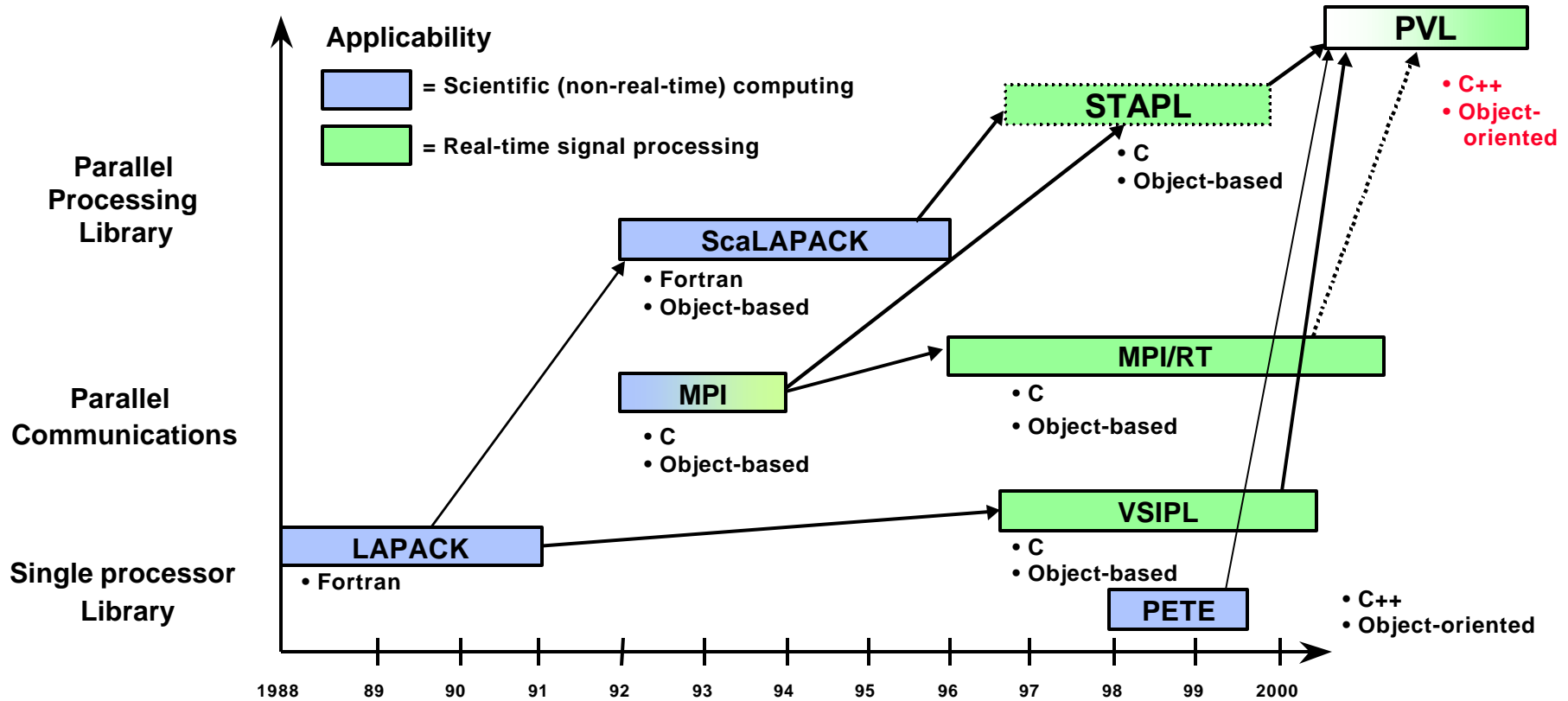
C++ Expression Templates in an Embedded, Parallel, Real-Time Signal Processing Library

Eddie Rutledge

***This work is sponsored by the US Navy, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and not necessarily endorsed by the United States Air Force.**



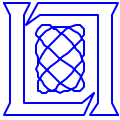
Evolution of Parallel Processing Libraries



PVL

- Collaboration between Lincoln and Lockheed Martin
- Component of AEGIS Common Signal Processor (CSP) Software Application Program Interface (API)
- Combines VSIPL API & parallel constructs from STAPL
- Portable, high performance, standardized signal processing library for real-time array signal processing

LAPACK = Linear algebra package
MPI = Message-passing interface
MPI/RT = MPI real-time
ScaLAPACK = Scalable LAPACK
VSIPL = Vector, Signal, and Image Processing Library
STAPL = Space-Time Adaptive Proc. Library
PETE = Portable Expression Template Engine



C Vs. C++ Code Simplicity

C (Functional Approach)

```
for (i=0;i<rows;i++)  
  for (j=0;j<cols;j++)  
    A[i][j]=B[i][j]+C[i][j];
```

C (Object Based)

```
vsip_vadd_f(&B,&C,&A);
```

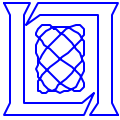
C++ (Object Oriented)

```
A=B+C
```

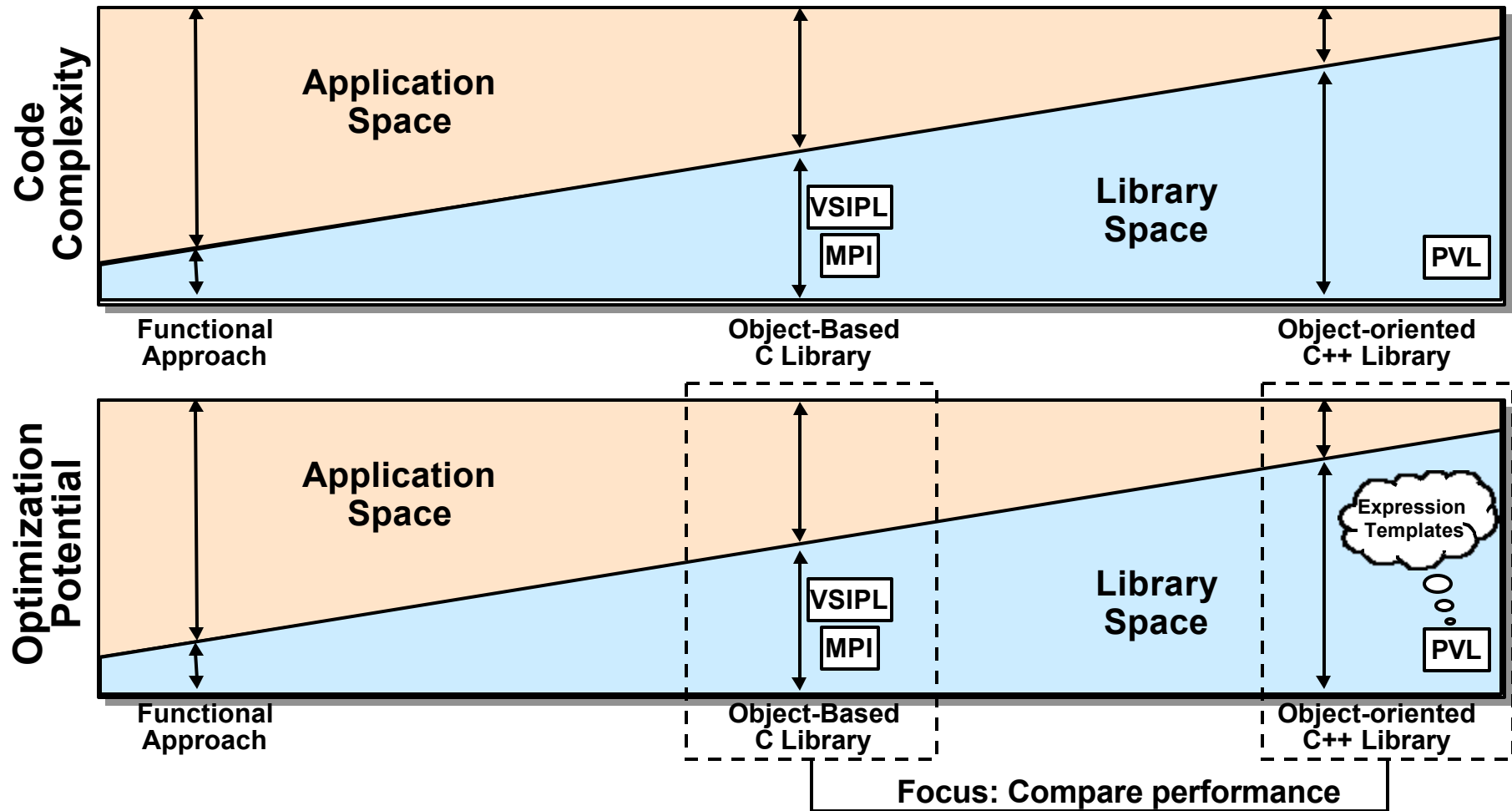
More intuitive translation from linear algebra to code

Performance concerns

We examine C++ expression templates as a way to increase code performance



C vs. C++ Performance

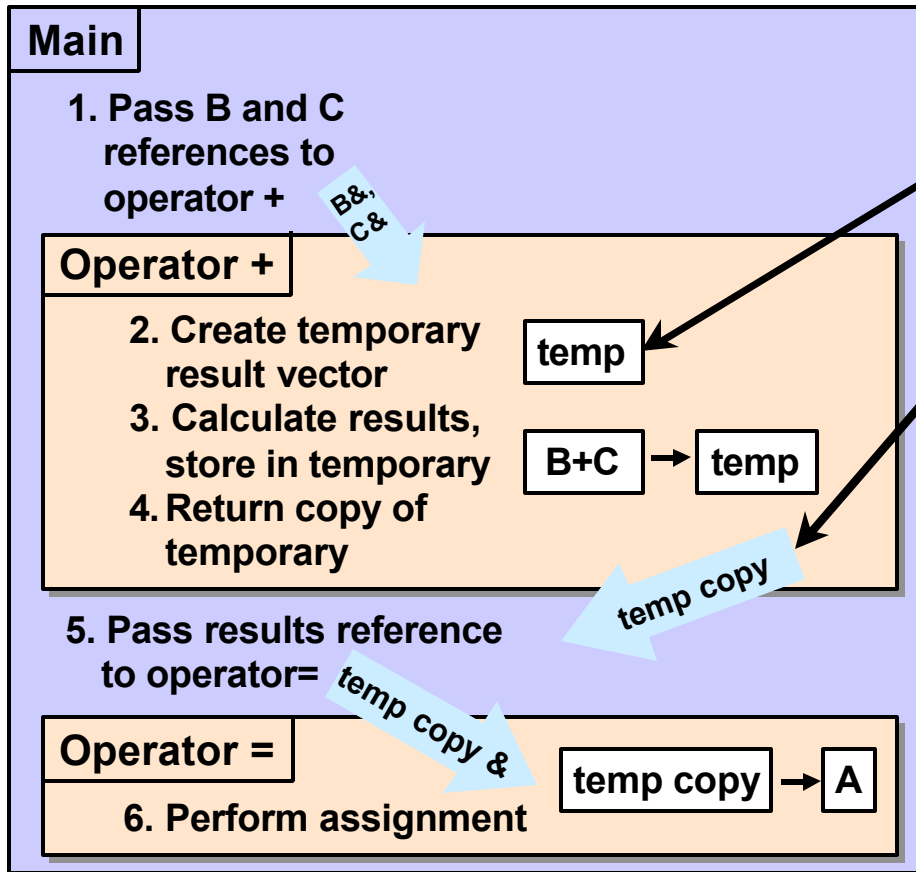


We compare the performance of the object-oriented C++ approach (PVL) with the performance of the object-based C approach (VSIPPL/MPI)



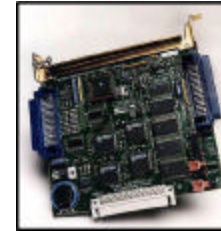
Typical C++ Operator Overloading-Overhead

Example: $A=B+C$ vector add



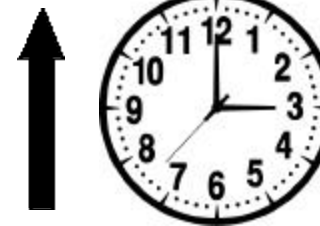
2 temporary vectors created

Additional Memory Use

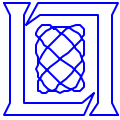


- Static memory
- Dynamic memory (also affects execution time)

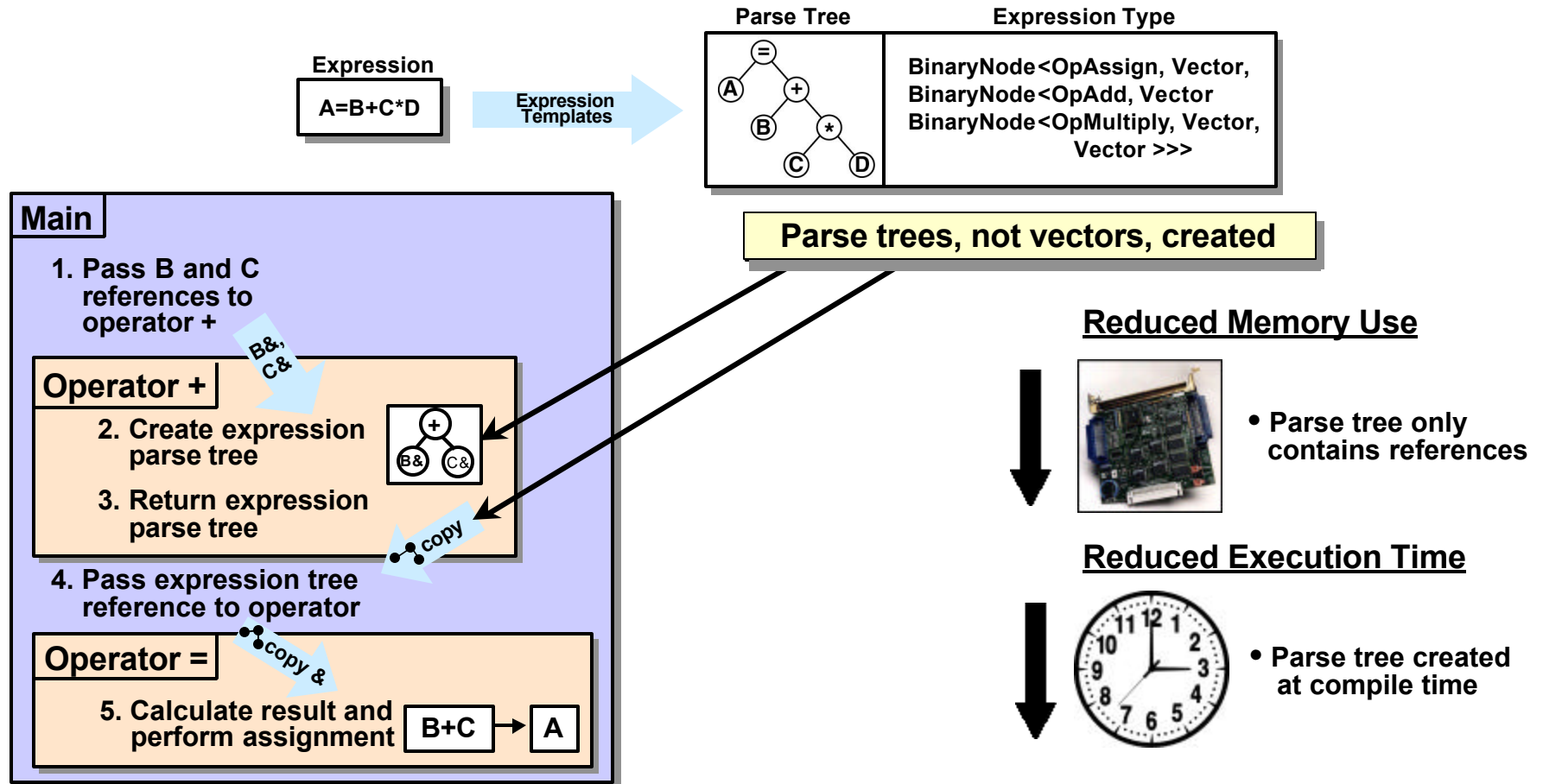
Additional Execution Time



- Time to create a new vector
- Time to create a copy of a vector
- Time to destruct both temporaries



C++ Expression Templates and PETE

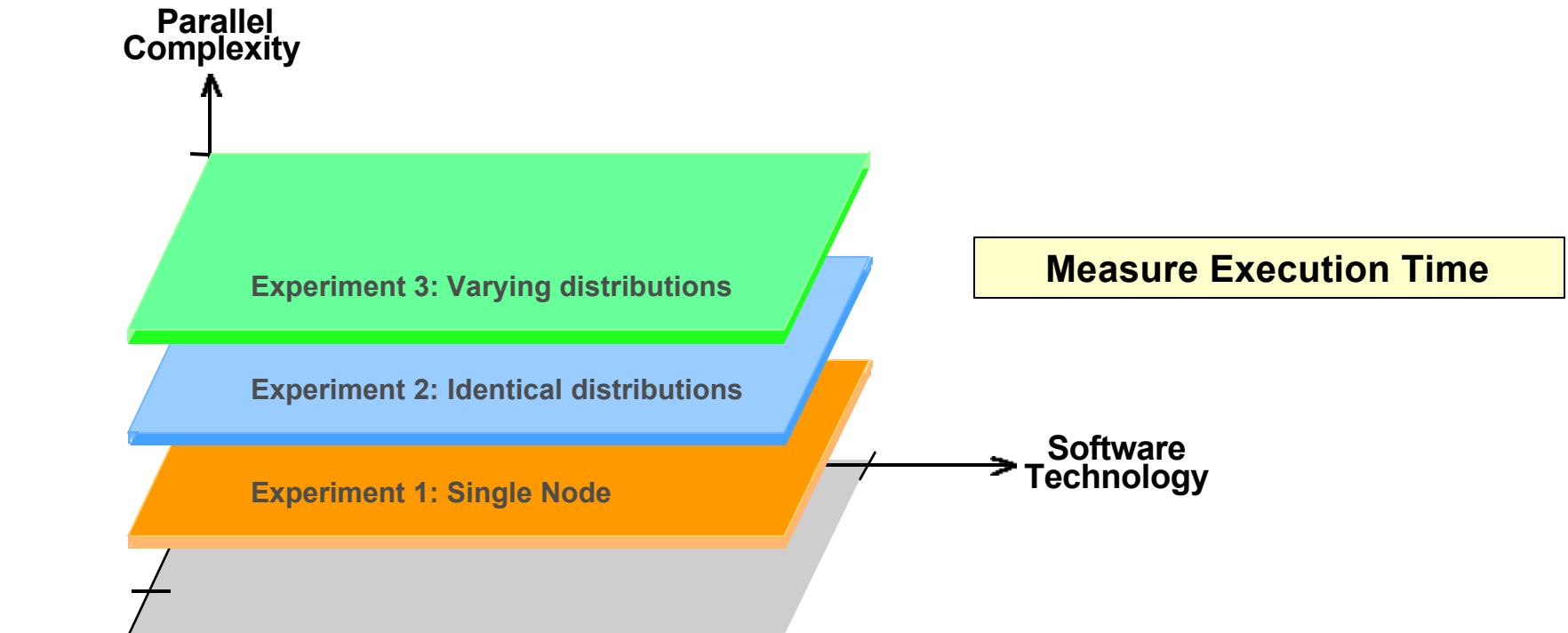


- PETE, the Portable Expression Template Engine, is available from the Advanced Computing Laboratory at Los Alamos National Laboratory
- PETE provides:
 - Expression template capability
 - Facilities to help navigate and evaluating parse trees

PETE: <http://www.acl.lanl.gov/pete>

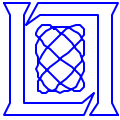


Experiments

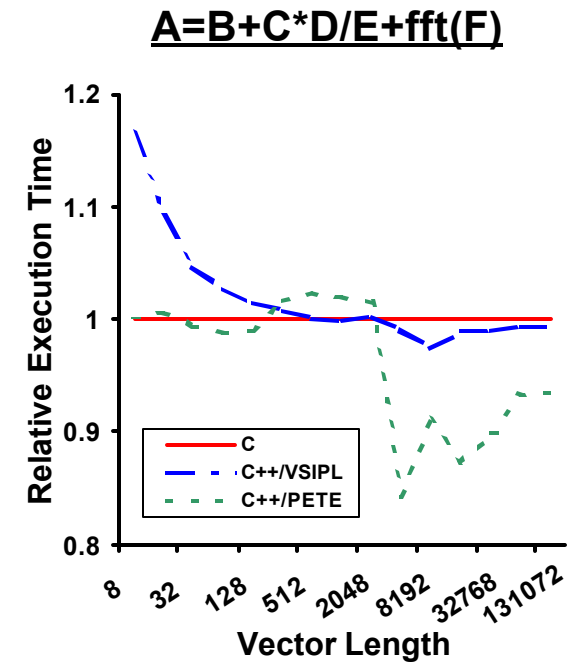
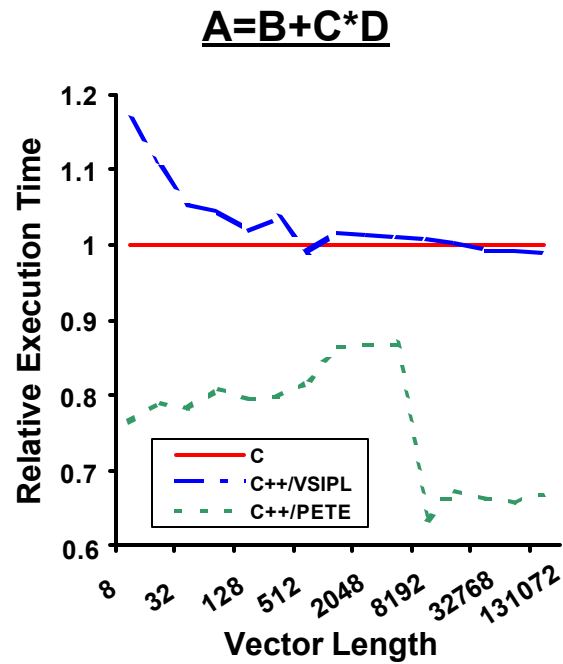
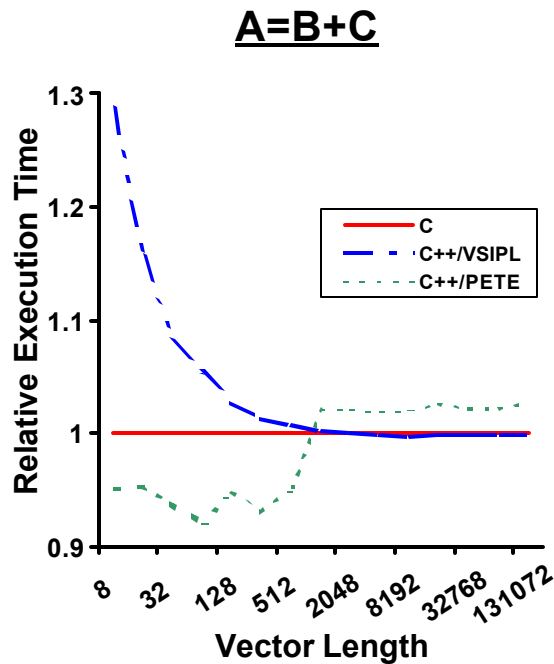


Software Technology

C	C++/VSIPL	C++/PETE
C MPI VSIPL	C++ PVL - MPI - VSIPL - PETE - Expression templates	C++ PVL - MPI - PETE - Expression templates - Evaluation



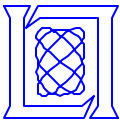
Experiment 1: Single Node



- Platform: Linux PC
- Element-wise multiply and divide

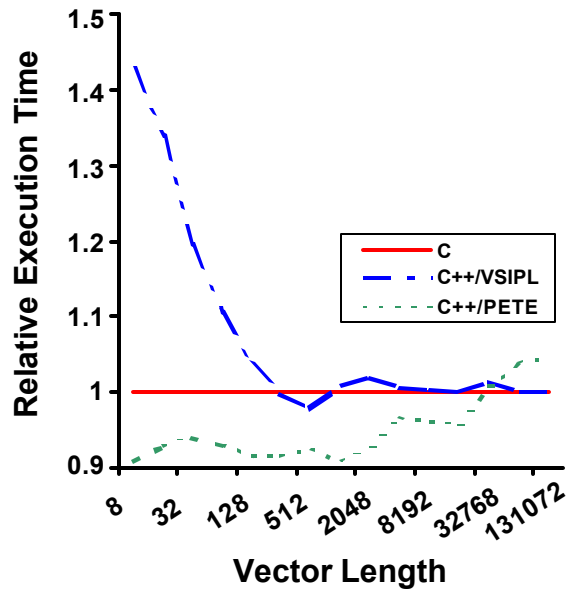
Relative overhead of C++/VSIP vs. C is small when expression templates are used

For longer chained expressions of element-wise operations, C++ with PETE element-wise evaluation outperforms other implementations

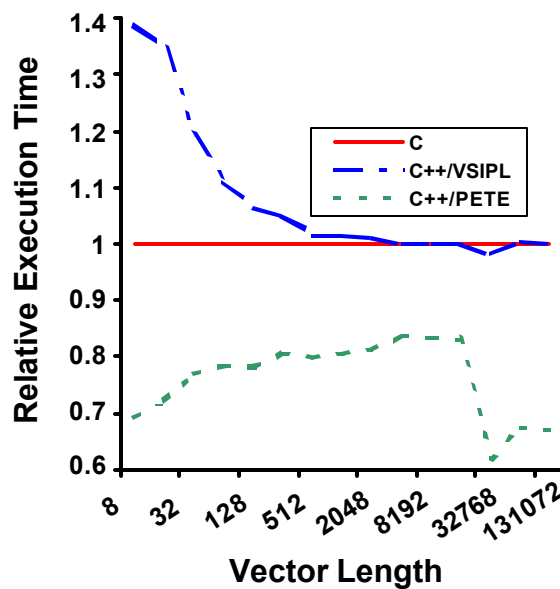


Experiment 2: Identical Distributions

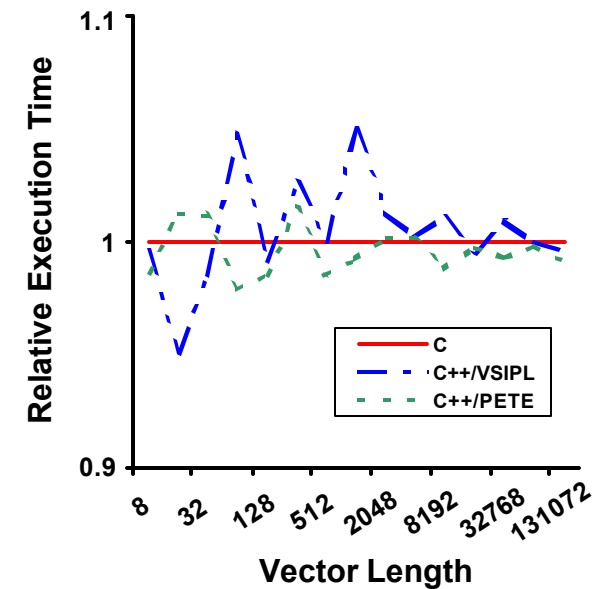
$$A=B+C$$



$$A=B+C*D$$



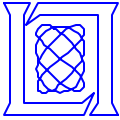
$$A=B+C*D/E+fft(F)$$



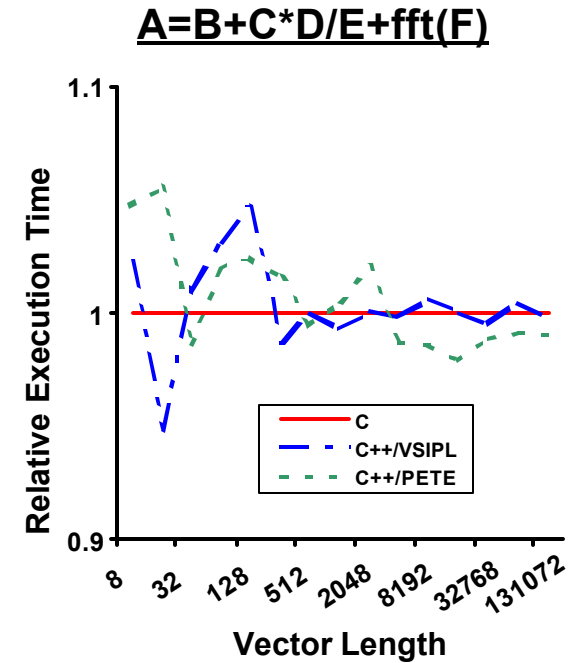
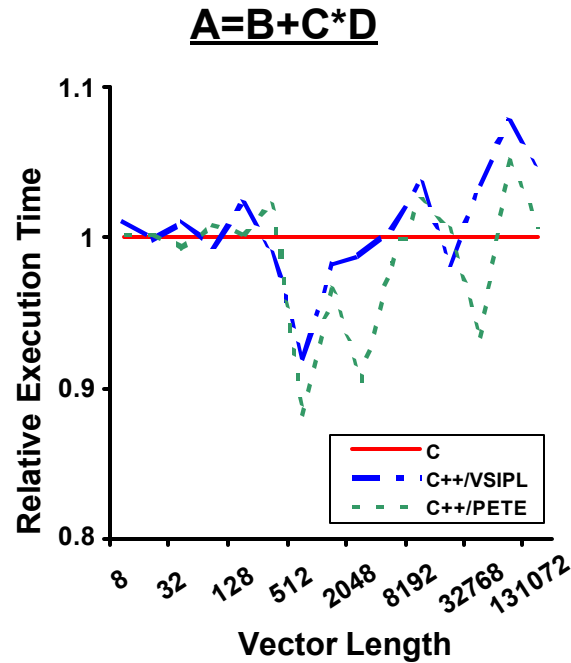
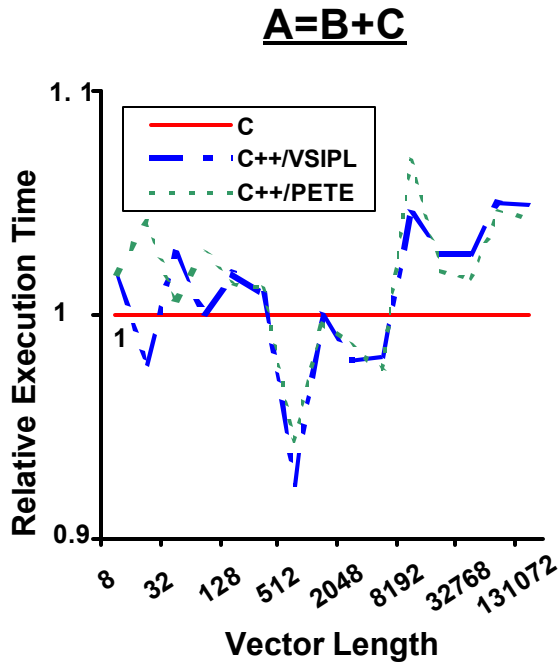
- Platform: 4 node Linux cluster
- Element-wise multiply and divide

Similar performance to single node case

Relative differences in $A=B+C*D/E+fft(F)$ are smaller because of communication required in (unoptimized) FFT

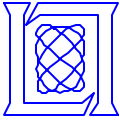


Experiment 3: Different Distributions



- Platform: 4 node Linux cluster
- Element-wise multiply and divide
- **A** distributed on 2 nodes, **B**, **C**, **D**, **E** and **F** distributed on 4 nodes

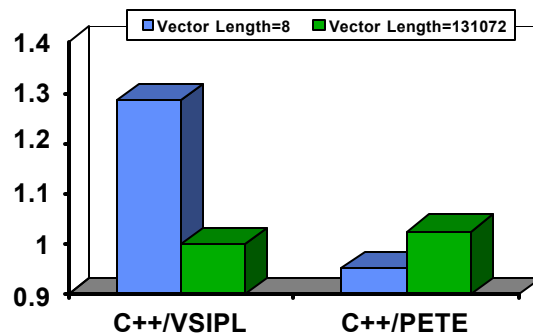
Relative differences between the 3 approaches are much smaller because of communication required in all cases



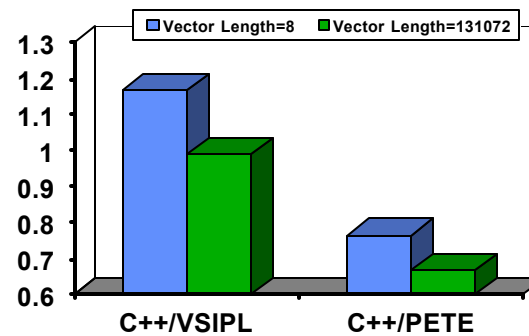
Results Summary

Execution time relative to C/VSIPPL implementation

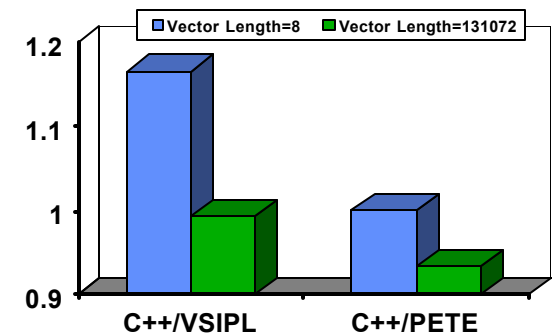
Single Node: A=B+C



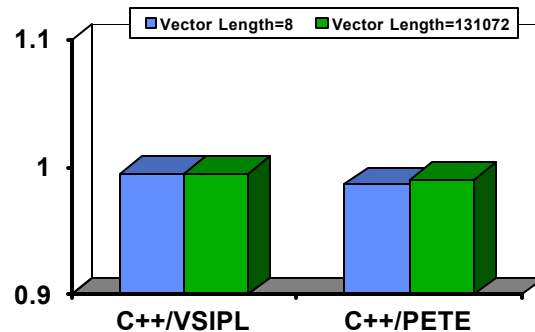
Single Node : A=B+C*D



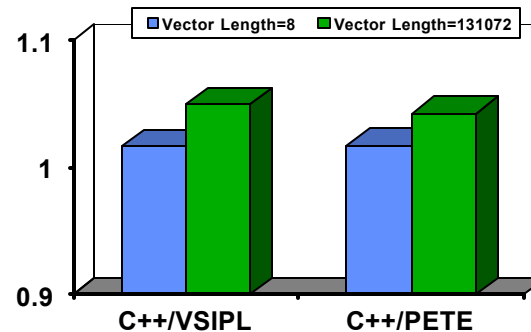
Single Node : A=B+C*D/E+fft(F)



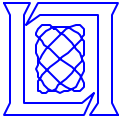
Identical Distributions: A=B+C*D/E+fft(F)



Different Distributions: A=B+C*D/E+fft(F)



- C++/VSIPPL- Small overhead compared to C
- C++/PETE- Significant advantage for chained expressions
- Relative differences largest where no communication is required



Future Optimizations

**2 optimizations made possible by expression templates
not possible with typical C++ operator overloading**

Minimize Op Count

Example: matrix-matrix multiply

$A = B \times C \times D \times E \times F$

B-30x35
C-35x15
D-15x5
E-5x10
F-10x20

Default ordering: 25,500 scalar multiplies
 $A = (((B \times C) \times D) \times E) \times F$

Optimal ordering: 11,875 scalar multiplies
 $A = (B \times (C \times D)) \times (E \times F)$

Minimize Communication

Example: $A = B + C$

- A and C are on processor 1
- B is on processor 2

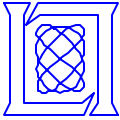
Typical C++ operator overloading solution
could involve unnecessary communication
of C to processor 2, then results to
processor 1

Optimal solution: Move B's data to
processor 1, then evaluate

Both optimizations require knowledge of the entire expression



Expression templates



Conclusions

- **C++ is preferable to C for high--performance signal processing**
 - More direct translation from linear algebra to code
 - Use of C++ expression templates results in similar performance
- **PETE element-wise evaluation is preferable in some cases to mathematical libraries such as VSIBL which necessitate the creation of temporary objects. Example: $A=B+C*D$**
- **Expression templates allow optimizations not possible with typical C++ operator overloading**
 - Minimize operation count. Example: $A=BxCxDxExF$
 - Minimize communication. Example: $A=B+C$

Acknowledgements

- **PETE was obtained from Los Alamos National Laboratory's Advanced Computing Laboratory
Lockheed Martin**

Phil Barile
Nathan Doss
Mary Frances Caravaglio
Michael Iaquinto

Jane Kent
Mike Lontoc
Rathin Putatunda
Roelan Teachey

MIT Lincoln Laboratory

Jim Daly
Jim Demers
Dan Drake
Hank Hoffmann
Jeremy Kepner
James Lebak

Jan Matlis
Patrick Richardson
Eddie Rutledge
Glenn Schrader
Ben Sadoski