Prior Art for Linear Algebra

November 22, 2018

Contents

1 Brief History of Linear Algebra Libraries in C++

1.1 Object-Oriented Numerics in the '90s

Linear algebra libraries in C++ at first evolved from the general category of "object-oriented programming." For example, the First annual Object-Oriented Numerics Conference (OON-SKI) took place in 1993[1]. Rogue Wave, a c++ compiler company, sponsored OON-SKI, so the conference logically focuses on work in C++. "Numerics" here means "numerical computation" or "scientific computing": computations on large arrays of floating-point numbers, doing things like discretizing differential equations or statistical data analysis [2]. The introduction to the corresponding journal special issue described the phenomenon of "object-oriented numerics" as follows:

> ... [W]e are observing the emergence of a subdiscipline: the use of object-oriented techniques in numerics. But what we are really seeing is something even more profound: finally rejoining of scientific computing with the science of computers. Traditionally, programming has been done by engineers, physicists and mathematicians with little or no training in computer science. Now, however, we are seeing an infusion of ideas coming from computer science world into the scientific computing world, brining along modern ideas on how to structure complex numerical code. Object-oriented techniques is merely one of many such ideas [VC93].

The introduction talks about the issues like needing to teach compilers how to fuse loops and avoid temporaries when doing overloaded-operator arithmetic on arrays in c++. It also shows the existence of C++ libraries for a variety of applications, including a library of multidimensional arrays, and the integrations of C++ with distributed-memory parallel computation.

One can see the explosion of interest in "object-oriented numerics" by the large variety of conferences that sprang up in the mid-90's [3]. Interest died down later, but this perhaps reflects the stage in scientific software developer where users came to accept externally developed libraries as part of their applications. In the U.S., this may correspond to the Department of Energy's ASCI program the resulting increase in complexity and fidelity of simulations. Simultaneously, the "killer micros" started to make traditional vector computing obsolete, and new distributed-memory parallel computers emerged. This may have influenced a switch from FORTRAN to other programming languages. Some newer computers did not have Fortran compiler, or required use of non-Fortran languages for best performance.

1.2 Templates

C++ had a reputation for poor performance compared with FORTRAN. Even developers willing to write C++ in "numerical" codes considered it better to use C++ has a high-level coordination

[1] `https://www.hindawi.com/journals/sp/si/250702/`

[2] Authors use adjectives like "numerical" to describe scientific and engineering computation.

[3] See e.g. `http://www.math.unipd.it/~michela/OP.htm#conferences`

language, and use a lower-level language like C for tight loops[Arg+97]. Developers saw C++ templates, in particular expression templates, as an optimization technique that could close the performance gap. Expression templates would let developers write compact, abstract code that "looks like math", yet optimizes by fusing loops and avoiding temporaries. For example, the Dr. Dobbs article on Blitz++ [4] focuses on expression templates for vector operations[5]. Developers also recognizes the cost of virtual method calls in C++, especially in inner tight loops, and used templates to reduce the cost of polymorphisms. For example, the Bernoulli Generic Matrix Library used the "Barton-Nackman trick"[BN94], a special case of "Curiously Recurring Template Pattern", to turn dynamic polymorphism into static polymorphism [6].

Early libraries that relied on templates suffered due to incomplete compiler implementations. For example, Blitz++'s installation process exercises the compiler to test language feature compliance. Its User's Guide recommends that if the compiler "doesn't have member templates and enum computations, just give up" [7]. A comparable library, POOMA (Parallel Object-Oriented Methods and Applications)[8], pushed the boundaries of what the available C++ compilers could handle. Chris Luchini, a POOMA developer, recalls that the project exposed many compiler bugs [9]. Many compilers lagged behind the C++ standard, only implemented a subset of features and generated slow code[MPS00].

Software for scientific computing may need to build with several different compilers and run on different kinds of hardware. Lack of consistently complete implementations of templates challenged portability requirements and restricted adoption. For example, in the Trilinos software project, a requirement to support a C++ compiler with incomplete template support drove the project to forbid templates in its foundational linear algebra library, Epetra [10].

1.3 POOMA

The POOMA (Parallel Object-Oriented Methods and Applications) project was most active 1998-2000. POOMA's goal was to support structured grid and dense array computations. As per Chris Luchini's oral history [11] and POOMA's documentation, the team had a particular interest in SGI Origin shared-memory parallel computers. POOMA shares features with more recently linear algebra libraries, such as polymorphism on storage layout and parallel programming model, so it is worth studying for historical lessons.

POOMA's main data structure is Array. Array has three template parameters: the rank (the number of dimensions), the entry type (e.g. double), and the "engine". Engines are about storage of data. They correspond somewhat to the Accessor policy in the mdspan multidimensional array proposal [Edw+]. An engine implements access of entries. Entries could actually exist in some storage somewhere, or be computed from indices and not actually stored. Engines also describe parallel distribution somewhat – e.g. through the MultiPatch engine [12].

POOMA's Internals and Ranges let users construct possibly strided multidimensional index ranges. These features let users write very general indexed loops, like in the ZPL programming language

[4] See e.g., the 2005 version of the Blitz++ 0.9 User's Guide: `http://physik.uni-graz.at/~crg/Programmierkurs1112/pdfs/blitz.pdf`.

[5] `http://www.drdobbs.com/cpp/scientific-computing-c-versus-fortran/184410315`

[6] In [MPS00], authors cite [Vel00]

[7] Blitz++ 0.9 User's Guide, Section 1.4.3.

[8] `http://www.nongnu.org/freepooma/tutorial/introduction.html`

[9] Oral history, collected by Trilinos developer Mark Hoemmen circa 2017-18.

[10] See [9]

[11] See [9]

[12] `http://www.nongnu.org/freepooma/tutorial/tut-04.html`

[Cha+98]. However, POOMA users had to work a bit harder on distributed-memory parallel systems, to expose “guard regions” with redundant storage on process boundaries [13].

The POOMA project had to “discover” experimentally how C++ templates work, and develop their own idioms. For example, the developers learned that C++ doesn’t permit templating on return type and then deducing the return type [14]. As mentioned above, POOMA developers also had to explore the limits of compiler correctness and performance.

1.4 C++ for More Radical Optimizations

As experience with C++ templates increased, some developers applied them to more radical code optimizations. For example, the Bernoulli Generic Matrix Library used templates to generate optimized sparse matrix codes from a high-level specification [15]. Bernoulli used a kind of relational algebra (described in detail in the PhD dissertation) that is somewhat analogous to the Ranges proposal [NCB18], in that it gives users a general way to describe operations over sequences, while optimizing by avoiding storage of temporary intermediate sequences.

1.5 Lessons Learned from Efforts in Other Programming Languages

Dongarra [16] gives an oral history of standardization of popular Fortran linear algebra libraries, including EISPACK and LINPACK. LINPACK came later. Here is a longer quote by Dongarra explaining LINPACK’s choice to rely on the BLAS:

> Since linear systems have perhaps a broader impact, LINPACK was going to a wider audience, and we felt that it would have a larger acceptance. This package was designed at a time when the biggest computers available were the vector computers. The vector supercomputers were just coming onto the scene, and the package was designed with vector computers in mind, so the package was designed to rely on an underlying set of routines called the BLAS (the Basic Linear Algebra Subprograms). The BLAS are a set of kernels which form the computational core of LINPACK; they are the vector operations that are going to be done over and over again in the package. The BLAS were a set of standard routines which were formed right before LINPACK was really kicked off, and we made a decision to use them. [17]

In so far as possible, the project wrote one version of the algorithms for four different data types (two different real precisions and two different complex precisions), and generated Fortran code for each of the four data types from this “abstract” representation. It may make sense to talk about High Performance Fortran (HPF) [KKZ07; KKZ11]. The language had support for distributed-memory

[13] This essentially means that POOMA did not do implicit boundary exchange. High-performance computing experts like to expose and reify the parallel distribution, and any redistribution operations. This helps them avoid communication and data movement, and makes parallel synchronization semantics clear. This is also a bit of a reaction to High-Performance Fortran, where even copying from one array to another could require parallel synchronization.

[14] `http://www.nongnu.org/freepooma/tutorial/tut-03.html`

[15] [AMP00] By the time, I (Mark Hoemmen) encountered the Bernoulli project, it had abandoned C++ code generation in favor of OCaml (or some other ML derivative)-based code generation framework. My guess is that avoiding intermediate high-level C++ step improved run-time performance and avoided compiler correctness issues

[16] `http://history.siam.org/oralhistories/dongarra.htm`

[17] Id

parallel arithmetic operations on possibly distributed multidimensional arrays. Compare also to the 2-D block cyclic data distributions that ScaLAPACK[Bla+97] supports for dense matrices. 2-D block cyclic distributions include block and cyclic layouts, both 1-D and 2-D, as special cases.

1.6 Vector Spaces and Parallel Data Distributions

How can I tell if I'm allowed to add two vectors together, or multiply two matrices? Is it enough for their dimensions to be compatible? Mathematicians would point out that two vector spaces might still differ, even if they have the same dimension or are otherwise isomorphic. I can't add a coordinate in 3-D Euclidean space to a quadratic polynomial with real coefficients, just like I can't add meters to seconds. Like a physical unit, a vector space is a kind of "metadata." Equating all isomorphic vector spaces strips off their metadata.

A distributed-memory parallel data distribution is very much like a vector space. It takes an N-dimensional vector space and imposes a two-dimensional index $(p, I(p))$ on it, Here $p \in [0, P)$ is the parallel process index (the "rank", in MPI terms), and $I(p)$ is the set of indices that live on Process p. The only thing that would make this a finite-dimensional vector space is the field over which the entries of a vector are defined. Even if two vectors x and y have the same dimension N, if the two vectors have different parallel distributions, I can't add them together without communication. "Communication" here may mean different things on different parallel computers, but this generally means some combination of moving data between processors (either through a memory hierarchy, or across a network), and synchronization between processors. Communication is expensive relative to floating-point arithmetic [18]. It also affects the correctness, because it may introduce deadlock, depending on what surrounding code does. Thus, programmers like to see communication made explicit, even if it is hidden behind a convenient interface.

Linear algebra libraries can make this easier for programmers through the abstract language of vector spaces. For example, the library can let users construct and pass around a parallel distribution using the two-dimensional indexing structure $(p, I(p))$ mentioned above[19]. Users then create matrices and vectors using distribution objects created in this way. The library can forbid implicit arithmetic operations between different vector spaces, but can make data redistribution and/or "communicating" arithmetic operations explicit. Users can also get the data distributions out of a matrix or vector, and check themselves whether two different distributions are the same. It's easier to explain all this to users in the abstract language of vector spaces.

Even if I don't care about distributed-memory parallelism, I may still care about shared-memory parallelism (threads) and memory affinity.

1.7 A Matrix has Four Vector Spaces

Matrices fulfill two different roles. First, matrices are 2-D data containers. Their rows have a data distribution, and their columns do also. Second, I can do matrix-vector products with a matrix. This makes a matrix a function from its domain vector space to its range vector space.

[18]

[19] TODO cite any of various "memory wall" sources

Bibliography

[Edw+] H. Carter Edwards et al. *mdspan : A Non-Owning Multidimensional Array Reference*.

[NCB18] Eric Niebler, Casey Carter, and Christopher Di Bella. *The One Ranges Proposal*. 2018.

[KKZ11] Ken Kennedy, Charles Koelbel, and Hans Zima. "The rise and fall of high performance Fortran". In: *Communications of the ACM* 54.11 (2011), pp. 74–82.

[KKZ07] Ken Kennedy, Charles Koelbel, and Hans Zima. "The rise and fall of High Performance Fortran: an historical object lesson". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM. 2007, pp. 7–1.

[AMP00] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. "A framework for sparse matrix code synthesis from high-level specifications". In: *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. 2000, p. 58.

[MPS00] Nikolay Mateev, Keshav Pingali, and Paul Stodghill. *The Bernoulli Generic Matrix Library*. Tech. rep. Cornell University, 2000.

[Vel00] Todd Veldhuizen. "Techniques for scientific C++". In: *Computer science technical report* 542 (2000), p. 60.

[Cha+98] Bradford L Chamberlain et al. "The case for high-level parallel programming in ZPL". In: *Ieee computational science and engineering* 5.3 (1998), pp. 76–86.

[Arg+97] Erlend Arge et al. "On the numerical efficiency of C++ in scientific computing". In: *Numerical Methods and Software Tools in Industrial Mathematics*. Springer, 1997, pp. 91–118.

[Bla+97] L Susan Blackford et al. *ScaLAPACK users' guide*. SIAM, 1997.

[BN94] John J Barton and Lee R Nackman. *Scientific and Engineering C++: an introduction with advanced techniques and examples*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[VC93] Al Vermeulen and Margaret Chapman. "OON-SKI: an introduction". In: *Scientific Programming* 2.4 (1993), pp. 109–110.