From Research Prototype to Production Software: The Lessons of the SLATE Project

2019 Collegeville Workshop on Sustainable Scientific Software (CW3S19)
St. John's University, Collegeville, MN, July 22-24, 2019

Jakub Kurzak Mark Gates Ali Charara Jack Dongarra

One of the main questions of scientific software development is how to produce high-quality code while conducting research and heavy prototyping. The Software for Linear Algebra Targeting Exascale (SLATE) project is a perfect example of a software package that had to start as a research prototype and is gradually evolving toward production-quality code, with the final goal of matching or outmatching the robustness of its venerable ancestors, LAPACK and ScaLAPACK.

Did ScaLAPACK really need a replacement?

Although this effort is unprecedented within the Exascale Computing Project (ECP), development of a ScaLAPACK replacement from the ground up was inevitable, due to the fact that ScaLAPACK was designed in the early 90s. Specifically, is was created as a response to the rise of distributed-memory systems and the message-passing paradigm, represented by the Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI) standards. The main design principle of ScaLAPACK was the preservation of the algorithmic layer of LAPACK with its revered numerical properties. The main innovation, and the cornerstone of its architecture, was the 2D block-cyclic (2DBC) matrix layout, providing load balancing and asymptotic scaling properties. Unfortunately, an array of design choices doom ScaLAPACK as a contender for exascale:

Implementation Language: The top, algorithmic layer of ScaLAPACK is largely inherited from LAPACK and, for the most part, is implemented in FORTRAN 77. This has a profound impact on many design choices in the package.

Fork–Join Execution: ScaLAPACK follows the fork-and-join programming model, with parallelism expressed mostly in the layer of Parallel Basic Linear Algebra Subprograms (PBLAS), and heavy reliance on blocking, collective MPI operations. On modern systems ScaLAPACK is punished harshly by network latencies and Amdahl's sections.

Hardcoded Memory Layout: The 2D block-cyclic layout is hardwired into ScaLAPACK. It only allows for uniform distribution of the matrix to a $P \times Q$ process grid using a fixed blocking factor NB. Access to submatrices requires the use of stride, which adversely affects caches and complicates copies. Also hardwired is column-major storage, which can obliterate the performance of certain operations on both CPUs and GPUs. And finally, the 2DBC layout is not easily convertible to other layouts without making a copy of the data.

Extremely Memory-Conservative Yet Wasteful: ScaLAPACK takes the stance of not allocating internally any substantial amount of memory. All necessary workspace memory has to be allocated by the user and ScaLAPACK is designed to cope with insufficient amounts by sacrificing performance. Also, ScaLAPACK implements in-place matrix transformations, even when out-of-place alternatives exist and provide increased parallelism and potential for much higher performance. On the other hand, ScaLAPACK stores triangular and symmetric matrices as full matrices, essentially wasting half of the storage.

It is basically impossible to consider GPU acceleration within the design constraints of ScaLAPACK. The futility of such attempts is easily demonstrated by the exercise of combining ScaLAPACK with multithreaded BLAS, which is known to produce inferior performance compared to ScaLAPACK running with one MPI process per core.

Why does SLATE have to be an evolving prototype?

SLATE had to start as a prototype because it is a radical departure from the old wisdom of ScaLAPACK, most notably abandoning the legacy 2DBC matrix layout in favor of tile layout, and abandoning the reliance on the standard BLAS in favor of batched BLAS. Although by now the architecture has solidified to some extent, the development is still marked by intensive research and heavy prototyping. This is mainly due to the fluid condition of the underlying software stack, as outlined by the following points:

No unified model for programming distributed-memory, multi-GPU systems: Some solutions drift towards the cache model, an example being NVIDIA's managed memory. Some solutions drift towards the distributed-memory model, an example being the NVIDIA Collective Communications (NCCL) library. Automated dataflow schedulers, such as PaRSEC, Legion, or StarPU are still considered research prototypes, and MPI+OpenMP+X seems to be the prevailing model for the time being, X being either NVIDIA CUDA or AMD HIP.

No standardized solution for node-level memory coherency: When considering node-level memory consistency, there is a couple of alternatives: OpenMP/OpenACC compiler directives, OpenMP/OpenACC runtime APIs, CUDA runtime API, and CUDA managed memory. NVIDIA's managed memory is the closest it comes to a complete solution. It offers fully automated coherency protocol with support for replication and prefetching. One major problem is that it is page-based, which is too big a granularity in many cases. Another is that it is proprietary, and in particular not currently supported by AMD's hardware.

No standardized solution for programming GPU kernels: OpenCL basically faded away, and OpenACC is generally not considered a performance champion for GPU kernel development. NVIDIA's CUDA is the go-to solution for the development of high-performance kernels, and AMD's HIP is basically identical to CUDA. Still, neither of them has the status of an actual standard, and it would be a bit of a stretch to call any of them a de facto standard just yet.

OpenMP tasking does not mix well with MPI: To start with, MPI does not really mix well with multithreading. The MPI standard introduced thread safety fairly recently, and it usually comes at the cost of performance. What's even worse is that in general MPI cannot be safely mixed with OpenMP tasking. Extreme caution and careful workarounds are necessary—until MPI supports the notion of the `MPI_TASK_MULTIPLE` mode of operation [4].

OpenMP tasking does not mix well with GPU APIs. OpenMP tasks are asynchronous by definition. For multi-GPU programming it makes perfect sense to place GPU calls inside OpenMP tasks. Unfortunately, GPU calls are not asynchronous by default and require the use of CUDA streams. We end up using two different (de)synchronization mechanisms that do not know about each other. Higher levels of parallelism demand more streams. At some point juggling multiple streams becomes tedious and error-prone.

Development of batched BLAS is lagging. Unlike ScaLAPACK, which relies heavily on the standard BLAS, SLATE relies on the batched BLAS for executing its most performance-critical operations. Unfortunately, standardization of batched BLAS is not gaining much traction. Also, the ongoing efforts mostly consider the C API, while a C++ API is required in SLATE. Very little is available in terms of actual high-performance implementations—a handful of routines from NVIDIA and virtually nothing from AMD.

Consider that ScaLAPACK developers already had a very well established BLAS standard at their disposal, and basically had to worry about one emerging programming parading—message passing. SLATE is being developed while the MPI standard is evolving, the OpenMP standard is evolving even faster, the only viable GPU programming systems are proprietary, and the batched BLAS standardization is lagging. Under such conditions, SLATE development relies on educated guesses about the technology trends and taking bets on some technologies materializing in the future.

How is SLATE sustaining the transformation?

C++

The role of the programming language is not to be understated. It is a firm belief of the development team that the project would not succeed—given its budget and manpower—if, for example, the C language was chosen for the implementation, rather than C++. It is hard to imagine the development of SLATE without the benefit of inheritance, templating, standard library containers, etc. Specifically, C++ is the main facilitator of SLATE's compactness.

Compactness

One of the main risk-mitigating solutions in SLATE is code compactness, which minimizes the volume of code potentially exposed to changes in the underlying programming paradigms, MPI and OpenMP. SLATE is incredibly compact compared to its predecessors implemented in FORTRAN (LAPACK, ScaLAPACK) and C (PLASMA, MAGMA). The compactness of SLATE is mostly due to the combination of three different techniques:

Templating of Precisions: SLATE handles multiple precisions by C++ templating, so there is only one precision-independent version of the code, which is then instantiated for the desired precisions. Operations are defined to apply consistently across all precisions. SLATE's BLAS++ component provides overloaded, precision-independent wrappers for all the underlying node-level BLAS, which SLATE's PBLAS are built on top of. Currently, the SLATE library has explicit instantiations of the four main data types: `float`, `double`, `std::complex<float>`, and `std::complex<double>`. The SLATE code should be able to accommodate other data types, such as half, double-double, or quad precision, given appropriate underlying node-level BLAS.

Templating of Execution Targets: Parallelism is expressed in SLATE's computational routines. Computational routines solve a sub-problem, such as computing an LU factorization (`getrf`), or solving a linear system given an LU factorization (`getrs`). In SLATE, these are templated on target (CPU or device), with the code typically independent of the target. The user can choose among various target implementations. In the case of accelerated execution (`Target::Devices`), the updates are executed as calls to batch `gemm`. In the case of multicore execution, the updates can be executed as:

- a set of OpenMP tasks (`Target::HostTask`),
- a nested `parallel for` loop (`Target::HostNest`), or
- a call to batch `gemm` (`Target::HostBatch`).

Handling of side, uplo, trans: The classical BLAS take parameters such as `side`, `uplo`, `trans` (named "`op`" in SLATE), and `diag` to specify operation variants. Traditionally, this has meant that implementations have numerous cases. The reference BLAS has nine cases in `zgemm` and eight cases in `ztrmm` (times several sub-cases). ScaLAPACK and the PLASMA likewise have eight cases in `ztrmm`. In contrast, by storing both `uplo` and `op` within the matrix object itself, and supporting inexpensive shallow copy transposition, SLATE can implement just one or two cases and map all the other cases to that implementation by appropriate transpositions. For instance, SLATE only implements one case for `gemm` (`NoTrans, NoTrans`) and handles all the other cases by swapping indices of tiles and setting `trans` appropriately for the underlying tile operations.

To illustrate the compactness of SLATE, consider the Cholesky factorization computational routine (`potrf`), used by the Cholesky driver. SLATE's `potrf` routine is approximately the same length as the LAPACK `dpotrf` code, and roughly half the length of the ScaLAPACK and MAGMA code (all excluding comments). Yet SLATE's code handles all precisions, multiple targets, distributed-memory and shared-memory parallelism, a lookahead to overlap communication and computation, and GPU acceleration. Of course, there is significant code in lower levels, but this demonstrates that writing driver and computational routines can be simplified by delegating code complexity to lower-level abstractions.

Layering

One of the main tools of software engineering is creating software layers with clearly defined responsibilities and interfaces. Over the course of SLATE's development, we also recognized the fact that different development models apply to different layers, and that the simple models need to be used when applicable.

Waterfall Where Applicable: SLATE has layers where the waterfall development model can be applied in the textbook manner. It is important to recognize that when such an opportunity presents itself it needs to be taken advantage of. In SLATE, such was the case with the BLAS++ and LAPACK++ components. First, a document was drafted describing both the requirements and the design [3]. Then implementation followed and was accompanied by the development of a test suite for verification. And currently, the layers are in a low-effort maintenance phase.

Spiral Where Applicable: SLATE also has layers which allow for a textbook application of the spiral model. The batched BLAS++ (BBLAS++) is the prime example. Initially, BBLAS++ went though the phase of requirements and design [1], then implementation, then validation [2]. Since then, however, it has been going through incremental stages of incorporating new features, in order to gradually absorb more and more complexity from the higher layers.

Agile/XP Where Necessary: Finally, there are layers in SLATE, which are in constant flux, due to ever-changing requirements. One example is the MOSI-inspired, node-level memory coherency protocol. Because of the inherent complexity of designing such a mechanism, the initial implementation only supported the absolute minimum functionality. Then it went through a rapid sequence of redesign and reimplementation stages, as new computational routines introduced an array of new requirements.

Main Points and Takeaways

- ScaLAPACK reached the end of its life cycle and had to be replaced.
- Exascale programming frameworks/paradigms are in a state of flux.
- The use of a modern programming language facilitates good software engineering.
- It is good to minimize the volume of code exposed to changing requirements.
- Different development methodologies apply to different software layers.
- Classic development models (waterfall, spiral) are still applicable.
- Agile/XP development is the response to fast-changing requirements.

References

[1] A. Abdelfattah, K. Arturov, C. Cecka, J. Dongarra, C. Freitag, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and P. Wu. SLATE working note 4: C++ API for batch BLAS. Technical Report ICL-UT-17-12, Innovative Computing Laboratory, University of Tennessee, December 2017. revision 01-2018.

[2] A. Abdelfattah, M. Gates, J. Kurzak, P. Luszczek, and J. Dongarra. SLATE working note 7: Implementation of the c++ API for batch BLAS. Technical Report ICL-UT-XX-XX, Innovative Computing Laboratory, University of Tennessee, June 2018. revision 06-2018.

[3] M. Gates, P. Luszczek, A. Abdelfattah, J. Kurzak, J. Dongarra, K. Arturov, C. Cecka, and C. Freitag. SLATE working note 2: C++ API for BLAS and LAPACK. Technical Report ICL-UT-17-03, Innovative Computing Laboratory, University of Tennessee, June 2017. revision 03-2018.

[4] K. Sala, J. Bellón, P. Farré, X. Teruel, J. M. Perez, A. J. Peña, D. Holmes, V. Beltran, and J. Labarta. Improving the interoperability between mpi and task-based programming models. In *Proceedings of the 25th European MPI Users' Group Meeting*, page 6. ACM, 2018.