**Dense Linear Algebra Motif (Cholesky Factorization)**

**Original Goals:**

The project began as an effort to see how far we could get toward a competitive HPL code. We wanted to test how several different aspects would play in the dense linear algebra arena:

* Index set functionality (ranges, domains, iterators) as support for block algorithms
* Global view with locality and distributions as support for distributed memory algorithms
* Task parallelism as support for asynchronous (e.g., look ahead) algorithms

It was, and still is, hoped that these characteristics of Chapel would lead to a much simpler tuned, asynchronous, LU code to serve as HPL in Chapel.

**What has been done:**

All of these ideas have been tested in the simpler context of the symmetric Cholesky factorization, where all have been shown to be important in dense factorization algorithms. We chose to do the Cholesky factorization rather than pivoted LU to avoid the complications of pivoting. The resulting codes are easier for non-linear algebra folks to understand and yet still exhibit most of the key properties.

*Standard Cholesky*

Because asynchrony was a goal, it was important to explain task dependency relationships to non-specialists. We began by implementing the three standard different orders of operations for a simple point / scalar Cholesky code. The “standard\_cholesky” directory contains three sets of codes, respectively, inner- and outer-product Cholesky and bordering Cholesky. From each of the scalar codes, I developed a one-dimensional (column slab) blocked code and then a two-dimensional blocked code. As they stand today, the 2D blocked codes are similar to LAPACK blocked codes. It is intended that they would be used together with a block 2D cyclic distribution to emulate the standard ScaLAPACK distributed memory codes, with the block size chosen internally to match the data distribution blocking factor.

The scalar codes look quite similar to scalar codes in Fortran or C, such as the factorization of a block in the LAPACK codes. The use of ranges has a mostly cosmetic effect at this level. The value of the index set functionality in Chapel is more evident and important in the block codes. In standard codes, the block size is referred to frequently and “off by one” errors on expressions like “lower + block\_size – 1” can occur in multiple locations. In these Chapel implementations, the references to block\_size are confined entirely to the iterator module. There are far fewer opportunities to drop or add an increment.

I would like to note that the writing of the three 1D block codes together required about a day of work, representing my first experience with iterators of this sort (but supported by Mary Beth Hribar’s earlier work). The three 2D codes were each developed from the corresponding 1D codes in less than ten minutes each!

Note – it is of course intended that these codes be used with a 2D block cyclic distribution. We are waiting for that to be developed. Without it, parallelism of any value is quite limited.

*Dataflow Cholesky*

Asynchrony was a goal and I decided to take it to its limit, since it seemed possible to do so in Chapel. It is straight-forward to express the task dependencies of the 2D blocked Cholesky factor, taking as tasks factoring a diagonal block, performing block solves on off-diagonal blocks and performing low rank modifications of blocks. The dataflow directory implements a “pure” dataflow 2D block Cholesky factorization using Chapel’s sync variables with extended memory semantics (full/empty bits). Each task has an associated counter of related tasks that must be completed prior to this task. Each task reduces the counters by one for any tasks that depend on its completion. Whenever a task reduces the counter of a dependent task to zero, it “schedules” that task by initiating a Chapel thread. Scheduling is left entirely to the Chapel run-time system.

This code is a substantial demonstration of the value of task parallelism in Chapel. The code is probably more complicated than future codes might be; Chapel plans to support atomic blocks, which are simpler to code. Sync variables are a bit closer to the metal than people may want to use. On the other hand, pure dataflow is a bit simplistic. Although the theory of what the best ordering for these operations is still a hard problem, heuristics weighted toward the critical path along the diagonal and sub-diagonal are what motivates the clumsy look ahead algorithms in HPL. Chapel does not today support priority queues or other scheduling options on its parallel tasks and no definitive plans are in place to do so.

Note – it is of course intended that this code be used with a 2D block cyclic distribution. We are waiting for that to be developed. Without it, parallelism of any value is quite limited. But the code is parallel. Locality is not taken advantage of yet, since the distributions available did not correspond to the native block structure.

*Elemental Cholesky*

By happenstance, as I began to develop an LU factorization from the Cholesky infrastructure, I received an intriguing paper from Robert van de Geijn at the University of Texas, describing a new distributed memory Cholesky factorization algorithm developed with his student Jack Paulson. The term “elemental” comes from their use of a 2D *unblocked* cyclic distribution, in which individual elements (entries) are distributed rather than blocks. Developing a Chapel version of this algorithm did not seem to represent an obvious step toward our eventual LU goal. In retrospect it was quite fortuitous that this algorithm appeared at this time.

van de Geijn has had a long running battle with the ScaLAPACK developers over whether 2D block cyclic data distribution is necessary and appropriate, or alternatively, overly complex and difficult. Without joining that battle, I will note that van de Geijn’s objections to the 2D block cyclic distribution were critical to our being able to implement a truly data parallel Cholesky code at this time. We don’t have a 2D block cyclic distribution, but we do have most of the simpler 2D cyclic distribution. That’s what was needed for the elemental Cholesky code. So we were able to go farther toward a true data parallel factorization code than we would could in the LU arena.

All of the major dense linear algebra codes known to me use a common model for the topology of the hosting parallel computer. Regardless of what the real topology is, all assume logically that the processors are arranged in a two-dimensional mesh / grid. That is really a least-common denominator for modern parallel systems, so the effort in implementation is mostly on tuning the communication steps. Those fall into several standard operations, typically gathers or broadcasts operating on the entire mesh, or on row or column submeshes. All end up storing some data redundantly.

The elemental algorithm has two other characteristics that represent challenges to Chapel in its current state. One is that it performs redundant *arithmetic*. A second is its use of submesh communications, where it performs an all-to-all exchange on one dimension of the mesh on data stored redundantly across the other dimension in the mesh.

Chapel today does not directly support redundantly stored data. There have been discussions as to how this may eventually be done, but the functionality is not yet defined. There has been even less discussion about redundant computation, so the elemental Cholesky code pushes on two different missing features in Chapel. I was able to sidestep these issues, mostly, by creating a true SPMD programming model within Chapel’s more general threaded model. By creating a single thread for each processor and assigning that thread to be local to a specific processor, I was able to create local (redundant) storage and redundant computation. It is to be seen whether the general Chapel model of parallelism can be used more cleanly as the language evolves.

The global memory model made expressing the communication steps almost trivial. Communication in this code is simply a set of simple assignment statements. However, there is no mechanism in today’s language to express communication between redundant copies of data. So the final communication step, which is exquisitely complicated in most languages, and extremely simple in Chapel, isn’t quite accurately represented in Chapel. The Chapel code uses the globally accessible version of an array rather than locally stored redundant copies. Hence, it has the wrong communications complexity.

Apart from this single discrepancy, the Chapel elemental code is a remarkably simple explication of the Paulson-van de Geijn algorithm.

*Elemental Factorization / strided or unsymmetric*

The usual languages for scientific computing assume that matrices have rows and columns numbered by one, starting at one (Fortran) or zero (C, etc.). That means that the rows and columns of a square matrix have the same set of indices. Chapel does not impose this restriction. However, with the exception of these two subdirectories, all of the codes in this evaluation assume that the row and column indices are identical.

I used the elemental factorization code as a basis to understand what changes were necessary to accommodate different index ranges for rows as for columns or to accommodate striding in either dimension. The *unsymmetric* directory contains code that assume unstrided ranges, but with different base indices. The *strided* directory contains codes that assume equal, but strided, ranges for both dimensions. (Combining the two characteristics is possible and inherits all the worst characteristics of both.) The lessons learned are described in a separate document. In many contexts striding or other differences in the two ranges will be handled naturally. This is especially true when operations on different entries are independent, leading to “forall” loops. In the factiorization case, where we must march along diagonals, and find “next” entries, the best solution is to use Chapel’s reindexing features, which allow the programmer to remap to the standard configuration used in other languages. This should have minimal effect on performance and avoids the programming challenges demonstrated in these two examples. Otherwise, handling striding or asymmetric ranges is a step backwards in programmer productivity.

*Elemental Factorization / block\_distribution*

This is a simple demonstration of the ease of changing from one distribution of data to another. The elemental factorization is designed so that locality can be exploited. But it works for other distributions, provided one doesn’t check that locality is what one expects in the cyclic case. Well, this demonstration is almost as simple as it’s supposed to be.

*Elemental Factorization / fully blocked*

One of the characteristics of the elemental factorization algorithm is that blocking factors for the sub-computations are independent of the non-existent data distribution block factor and of each other. The original elemental code was intended as proof of concept; I ignored blocking. The fully blocked version blocks each of the factor, solve and Schur complement steps with separate blocking factors. The changes are generally quite small, one of the virtues of Chapel. The only interesting aspect of developing this blocked version was the need to provide iterators counting off segments of strided ranges. I would have liked to have used the count (“#”) operator and thereby have avoided any explicit computations using the actual stride of the range. I was unable to do this in the way I intended because the count operator does not like incomplete subintervals as occur at the edge case with a less than full final block.

*Performance*

In general we have ignored performance issues in our evaluation of Chapel, mainly because the compiler is so immature. We did ask for help from the Chapel team in evaluating performance of the simple scalar factorization codes because they were so slow, running at perhaps 1% of the speed of equivalent Fortran codes. The performance subdirectory contains a small set of variations on the scalar outer product factorization code.

First, we should note that Chapel by default lays out memory in a row major fashion. I ignored this in my code development and wrote codes to make them as easy to compare to the standard (Fortran) codes from LAPACK and ScaLAPACK. So where memory orientation might make a difference, I coded as if Chapel, like Fortran, used a column major memory layout.

Chapel programmers will eventually be able to control memory layout through the distribution functionality. It will be trivially more difficult to work on column major memory layout as the default row major memory layout. This will be an important advantage to Chapel.

At present we don’t have that capability. So the performance directory includes variants on the outer product scalar Cholesky factorization oriented for row-major storage as well as the previous column-major storage form. There are several variations to test what permits or deters the current compiler from getting good run-time performance. In fact, there is one variant which runs at similar speed to Fortran when compiled with the --fast compiler option. So performance is possible, but getting it from the prototype compiler is perhaps more effort than it is worth.

Testing environment

I did not try to emulate the HPCC HPL testing environment. The HPL tests involve generating a random general square matrix, which is done in questionable fashion. The error bounds in HPL are also questionable. In any case, we need a *symmetric positive definite* (SPD) matrix, not a general square matrix.

This package generates a random SPD matrix by first generating a random general matrix and then forming the SPD matrix. The matrix is SPD by construction and is highly likely to be well-conditioned. We do not check the conditioning of and the error bound we use to evaluate success or failure does not depend on conditioning. We explicitly compute the matrix residual. The error bound is by Demmel, found as Theorem 10.5 in Higham's **Accuracy and Stability of Numerical Algorithms, 2nd Ed.**

The error bound is rigorous and applicable, unlike the HPCC HPL bounds. However, the pre- and post-computations both involve operations, just like the factorization. No attention has been paid to parallelizing this computation. As matrix-matrix multiplication, it is highly scalable, but its expense is considerable at scale.

**What is missing today from Chapel for HPL:**

To be able to move to a real HPL code, Chapel needs to have:

* More and better distributions. This includes of course a block 2D cyclic distribution. But the interfaces to distributions need to be standardized and there is a lot of work yet to be done to make it easier to develop distributions.
* Replicated data and means to replicate computation.
* Means to express communications among sub-teams of processors.
* Probably communications libraries for standard communications primitives. These are obvious optimizations to stand in place of global assignments for well-designed algorithms. They probably don’t have to require the memory copies of MPI.
* Working “local” clauses to help code run efficiently on local data.
* More control over runtime scheduling.
* Atomic statements.
* A more robust way to terminate processes besides a nuclear bomb of a “halt”.

**What still needs to be done with these codes:**

The 2D block standard Cholesky codes and the data flow codes depend on having a block 2D cyclic distribution. Once that is available, true parallel experiments can begin on these codes, especially the data flow code. In both cases, it will become important to decide how to control locality. In addition, there are a number of places where parallel “forall” loops are possible, but depend on writing parallel “leader/follower” iterations. This iterator form was not documented until the most recent language specification was released. Time constraints and the unavailability of the appropriate distributions deterred me from implementing any of these. Without them the full extent of parallelism possible in the code has not been expressed. There are comments in the code suggesting where leader-follower iterators should be used, but exercise due diligence in implementing these comments.

The 2D block standard Cholesky codes and the elemental code depend on facilities for replicating data and computation. It would be a useful experiment to rewrite the elemental code in the standard Chapel parallelism model once replication is really available. At this time, code length comparisons with ScaLAPACK become meaningful. (Note that these codes do not include the LAPACK/ScaLAPACK alternative of choosing whether to store the factor in the upper or the lower triangle. At first blush it would be simpler to cut from the latter than to build a second version in Chapel. But Chapel’s code is so simple, the latter alternative might be easier.)

Once this point is reached, it would make sense to tackle the LU factorization. It will be very interesting to see how best to represent pivoting, to see what features of Chapel address that.

Communications performance can be studied in either the Cholesky or the LU context. The elemental factorization would provide an interesting alternative test bed for that study.

Meshing dataflow with the standard codes is an eventual goal, because Chapel supports blocking and dataflow. But there are aspects of this that are still very much research and hard research at that.