Parallel Linear Algebra in Julia

Britni Crocker and Donglai Wei

18.337 Parallel Computing

12.17.2012

Table of Contents

Abstract

In this project, we implement two Parallel Tiled Linear Algebra Algorithms in Julia, Cholesky and QR decomposition. We follow the algorithm description in [1] and use “dlsym” and “ccall” functions in Julia to load the OpenBLAS dynamic library [2] and call BLAS kernels. From this foundation, we are able to implement serial and parallel tiled linear algebra algorithms in Julia. For parallel implementation, spawned tasks could be dynamically scheduled based on their dependencies to the returned references of previous tasks.

Introduction

2.1 Problem Statement

Numerical linear algebra plays an essential role at the center of many different applications within engineering and computational science. Due to its optimized implementation, the software package LAPACK and underlying BLAS routines became the ubiquitous choice for executing basic linear algebra methods. However, LAPACK is somewhat difficult for non-expert programmers, spurring the integration of this library with many high-level languages and computing environments geared specifically for scientific and engineering use. These software applications generally sacrifice performance for ease-of-use and therefore may not be appropriate for handling large amounts of data or running large-scale simulations. The new Julia programming language seeks to combine high performance with high-level usability, including intuitive support for parallel computing.

Our goal is to build a scalable distributed dense numerical linear algebra library in Julia. Specifically, we aim at implementing two algorithms: Cholesky and QR decomposition. First, we seek to implement the serial version of these numerical algorithms (as given in [1]) by exploiting Julia's ability to call C and Fortran libraries in order to directly use methods from the BLAS kernel. Our second task is to use the Directed-Acyclic-Graph (DAG) scheduler in julia to execute the code in parallel.

2.2 Background

SCALAPACK is the current standard library for performing parallel dense linear algebra operations built upon LAPACK. The Cholesky and QR decomposition implemented in SCALAPACK is designed as a high-level algorithm relying on basic block routines from the Basic Linear Algebra Communication Subprograms (BLACS), based on the Basic Linear Algebra Subprograms (BLAS). The matrix is conceptually split in blocks of columns, and the decomposition proceeds by updating the trailing submatrix at each step until all the subsequent panels are processed. The main drawback to the SCALAPACK approach is that synchronization points are required between operations, meaning parallelism only occurs within each phase (panel or update) expressed at the level of BLAS and BLACS.

To alleviate these bottlenecks, tile-based algorithms break the panel decomposition update of trailing submatrices into finer granularity. The update phase can now be initiated while the corresponding panel is still being factorized, which allows asynchronous execution models to hide the latency of access to memory. In this report, we use the high-level algorithm presented in [1]. In more recent work, implementations on different hardware (e.g. GPU) and variations of updating order and adaptive block size are explored.

Tile-based linear algebra algorithms are not naively parallelizable. While every step in the algorithm may only require a few tiles, each step often depends on the results of one or more previous steps. Because of this structure, these tile-based algorithms can be represented as directed acyclic graphs (DAGs) - directed graphs with the special property that it is impossible to start at some node n and return to that same node by following the directed edges [5]. Here, each node is one of the tile-based tasks that needs to be completed in the algorithm, and each directed edge represents the dependency of that task on the results of previous tasks.

2.3 Linear Algebra Algorithms

We focused on two matrix decompositions: Cholesky and QR. Both of these operations have tile-based serial algorithms, and the parallel versions are constructed by dynamically allocating tasks based on a DAG representation of the algorithm.

Cholesky Decomposition

The Cholesky decomposition of a symmetric positive definite matrix A determines the lower-triangular matrix L, where LL' = A.

The original algorithm is a modification of Gaussian elimination. It consists of repeatedly performing the decomposition of the lower-left submatrix. In the ith iteration, we perform $\mathbf{A}^{(i)} = \mathbf{L}_i \mathbf{A}^{(i+1)} \mathbf{L}_i^*$ to subtract $a_{i,i}$ from the corresponding row and column vector b_i and b_i^*. In the end, we have $L=L_1...L_n$.

$$\mathbf{A}^{(i)} = \begin{pmatrix} \mathbf{I}_{i-1} & 0 & 0 \\ 0 & a_{i,i} & \mathbf{b}_i^* \\ 0 & \mathbf{b}_i & \mathbf{B}^{(i)} \end{pmatrix}, \mathbf{L}_i := \begin{pmatrix} \mathbf{I}_{i-1} & 0 & 0 \\ 0 & \sqrt{a_{i,i}} & 0 \\ 0 & \frac{1}{\sqrt{a_{i,i}}}\mathbf{b}_i & \mathbf{I}_{n-i} \end{pmatrix},$$

Image from Wikipedia

Notice that the last step of the basic algorithm above requires matrix multiplication, which is inefficient. Alternatively, by writing out the element of L explicitly, we can find the recursive algorithm to directly obtain the value of elements in L.

$$
\begin{aligned}
\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}} &= \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix} \\
&= \begin{pmatrix} L_{11}^2 & & \text{(symmetric)} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix}
\end{aligned}
$$

$$
L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right), \qquad \text{for } i > j.
$$

Image from Wikipedia

QR Decomposition

The QR decomposition of a m×n matrix A, with m ≥ n, results in an m×m unitary matrix Q and the m×n upper triangular matrix R, where QR=A. As the bottom (m−n) rows of an m×n upper triangular matrix consist entirely of zeroes, it is often useful to partition R, or both R and Q:

$$
A = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = \begin{bmatrix} Q_1, Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1,
$$

Image from Wikipedia

where *R*1 is an *n*×*n* upper triangular matrix, *Q*1 is *m*×*n*, *Q*2 is *m*×(*m*−*n*), and *Q*1 and *Q*2 both have orthogonal columns.

One direct algorithm is a modification of Gram–Schmidt process. The QR decomposition of $A=(a_1,a_2,...,a_n)$ can be seen as making column vectors a_i orthogonal to each other through multiplication of an upper triangular matrix: $AR^{-1}=Q$. In specific, we define the projection of of column vector a onto coordinate basis e as: $\mathrm{proj}_{\mathbf{e}}\mathbf{a} = \frac{\langle \mathbf{e}, \mathbf{a} \rangle}{\langle \mathbf{e}, \mathbf{e} \rangle}\mathbf{e}$. Then:

$$
\begin{aligned}
\mathbf{u}_1 &= \mathbf{a}_1, & \mathbf{e}_1 &= \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|} \\
\mathbf{u}_2 &= \mathbf{a}_2 - \mathrm{proj}_{\mathbf{e}_1} \mathbf{a}_2, & \mathbf{e}_2 &= \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} \\
\mathbf{u}_3 &= \mathbf{a}_3 - \mathrm{proj}_{\mathbf{e}_1} \mathbf{a}_3 - \mathrm{proj}_{\mathbf{e}_2} \mathbf{a}_3, & \mathbf{e}_3 &= \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} \\
&\vdots & &\vdots \\
\mathbf{u}_k &= \mathbf{a}_k - \sum_{j=1}^{k-1} \mathrm{proj}_{\mathbf{e}_j} \mathbf{a}_k, & \mathbf{e}_k &= \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}
\end{aligned}
$$

$$Q = [\mathbf{e}_1, \cdots, \mathbf{e}_n] \qquad \text{and} \qquad R = \begin{pmatrix} \langle \mathbf{e}_1, \mathbf{a}_1 \rangle & \langle \mathbf{e}_1, \mathbf{a}_2 \rangle & \langle \mathbf{e}_1, \mathbf{a}_3 \rangle & \dots \\ 0 & \langle \mathbf{e}_2, \mathbf{a}_2 \rangle & \langle \mathbf{e}_2, \mathbf{a}_3 \rangle & \dots \\ 0 & 0 & \langle \mathbf{e}_3, \mathbf{a}_3 \rangle & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

However, due to stability issues in the approach above, in practice QR decomposition is realized through Householder reflections. Given a vector x, the Householder reflection is defined as: $\mathbf{u} = \mathbf{x} + \alpha \mathbf{e}_1$, $\mathbf{v} = \frac{\mathbf{u}}{\|\mathbf{u}\|}$, $Q = I - 2\mathbf{v}\mathbf{v}^T$. where Q is an m-by-m Householder matrix and $Q\mathbf{x} = (\alpha, 0, \cdots, 0)^T$. For QR Decomposition, we multiply *A* with the Householder matrix Q_1 that we obtain when we choose the first matrix column for x. This results in a matrix Q_1A with zeros in the left column (except for the first row):

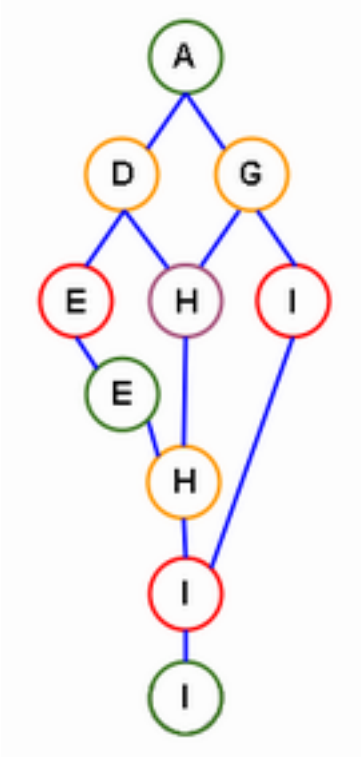
$$Q_1 A = \begin{bmatrix} \alpha_1 & \star & \dots & \star \\ 0 & & & \\ \vdots & & A' & \\ 0 & & & \end{bmatrix}$$

This can be repeated for A′ (obtained from Q_1A by deleting the first row and first column), resulting in a Householder matrix Q'_2. Note that Q'_2 is smaller than Q_1. Since we actually want it to operate on Q_1A instead of A′ we need to expand it to the upper left, filling in a 1, or in general:

$$Q_k = \begin{pmatrix} I_{k-1} & 0 \\ 0 & Q'_k \end{pmatrix}.$$

After t iterations of this process, $t = \min(m-1, n)$, $R = Q_t \cdots Q_2 Q_1 A$ is an upper triangular matrix. So, with $Q = Q_1^T Q_2^T \cdots Q_t^T$, $A = QR$ is a QR decomposition of A.

2.4 Parallelization:



As previously described, these tile-based algorithms can be described as a DAG. One example of such a representation is shown to the right (and a better one can be found in [3]) for the Cholesky algorithm. In this case, a matrix is split into 9 tiles (3x3), each represented by a letter A-I. The colors of the circles represent which of the four linear algebra steps is being executed: dpotrf, dtrsm, dsyrk, dgemm (see pseudocode in 3.2). Each blue line represents a dependency on a previous task (directed in this case from top to bottom).

All tasks in the same row do not depend on each other and therefore can be executed in parallel. A scheduler could use this DAG representation to dynamically assign tasks to computing nodes as parent tasks get completed, speeding up performance. In this small 3x3 example, 10 total tasks on tiles need to be completed, but there are only 7 rows of tasks in the DAG, so in an ideal case the parallel Cholesky calculation would take only 70% of the serial code execution time. However, the additional overhead of copying or communicating tiles from one node to another may reduce these gains or even cause the parallel code to run much slower.

Julia Implementation

3.1 Julia Language

The Julia programming language offers a number of convenient features for programming tile-based matrix decomposition with dynamic DAG-based scheduling. Julia allows us to directly call BLAS from a previously compiled shared library using ‘ccall’. It also takes care of a great portion of the dynamic scheduling - Julia’s spawn macro supports asynchronous execution and automatic selection of compute nodes. Finally, the high-level language has a gentle learning curve compared to other parallel computing software solutions.

We implemented the pseudocode from [1] in both serial and parallel versions. In the serial version, we focus on finding suitable kernels to use from OpenBLAS library, and in the parallel version we explore the DAG parallelization mechanism that Julia provides.

3.2 Serial version

Since [1] implements their own kernels, some of which are not in the OpenBLAS library, we need to understand the operation going behind the pseudo code so that we can find equivalent matrix/vector operations in OpenBLAS.

Cholesky Decomposition

```
for k = 1:num_tiles
   A[k,k] = dpotrf(A[k,k])
   for m = k+1:num_tiles
      A[m,k] = dtrsm(A[k,k],A[m,k])
   end
   for n = k+1:num_tiles
      A[n,n] = dsyrk(A[n,k],A[n,n])
      for m = n+1:num_tiles
         A[m,n] = dgemm(A[m,k],A[n,k],A[m,n])
      end
   end
end
```

Pseudo Code from [1]

dpotrf(A): standard Lapack function to perform the Cholesky decomposition for a symmetric matrix A=LL', and the output L overwrites A.

dtrsm(A,B): standard Lapack function to solve the linear equation AX=B, and the output X overwrites B.

dsyrk(A,B): standard Lapack function to update B := alpha*A*A' + beta*B, and here we need B: = -AA'+B

dgemm(A,B,C): standard Lapack function to update C:= alpha*op(A)*op(B) + beta*C, and here we need C := -A*B+C

In the end, we have the output L overwrites input matrix A.

QR Decomposition

```
for k = 1:num_tiles
   A[k,k], T[k,k] = dgeqrt(A[k,k])
   for m = k+1:num_tiles
     A[k,k], A[m,k], T[m][k] = dtsqrt(A[k,k],A[m,k],T[m,k])
   end
   for n = k+1:num_tiles
      A[k,n] = dormqr(A[k,k], T[k,k], A[k,n])
      for m = n+1:num_tiles
         A[k,n], A[m,n] = dssmqr(A[m,k],T[m,k],A[k,n], A[m,n])
      end
   end
end
```

Pseudo Code from [1]

dgeqrt(A): standard Lapack function to perform the QR decomposition for matrix A=QR=(I-VTV')R, and the output R and V overwrites A using the compact WY technique to accumulate Householder reflectors. We need extra memory to store T.

dtsqrt(A,B,C): not available in Lapack, so we substitute with Lapack function dtpqrt(A,B,C) and parameter L=0.

dormqr(A,B,C): standard Lapack function to apply the reflectors Q stored in A and B, calculated by dgeqrt to the tile C, such that C := QC

dssmqr(A,B,C,D): not available in Lapack, so we substitute with Lapack function dtpmqrt(A,B,C,D) and parameter L=0.

In the end, we have the output R overwrites input matrix A. It is cumbersome to carry out the calculation to recover the orthogonal matrix Q by taking outer product of all reflectors generated along the way. Experimentally, we find it faster to implement a naive paralleled dtrsm to solve the linear equation A=QR.

3.3 Parallel version

Dynamic, asynchronous scheduling in Julia can be achieved through the macro spawn. Under this framework, any task called with spawn can be completed at any time, in any order, relative to other spawned tasks. Unless manually specified, Julia will choose the processor to perform the task, and every spawn returns a remote reference to the output of the task. The spawn macro in Julia provides a powerful way to implement tasks that don't necessarily complete in the order you've written them. In our DAG representation, these spawned tasks represent the

graph nodes.

In Julia, scheduling of spawned tasks do not consider the references returned by other spawns or the relative placement of other spawns. However, scheduling does depend on the relation of that spawned task to other variables outside the spawns. We can exploit this feature to generate the directed edges of the DAG by explicitly making variables that refer to remote references returned by spawned tasks. In this way, spawned tasks can be forced to wait for returned values from other spawned tasks. A pseudocode example for Cholesky is given below:

```
for k = 1:num_tiles
    M_kk = M[k,k]
    M[k,k] = @spawn dpotrf(fetch(M_kk))
    for m = k+1:num_tiles
        M_mk = M[m,k]; M_kk = M[k,k]
        M[m,k] = @spawn dtrsm(fetch(M_kk),fetch(M_mk))
    end
    for n = k+1:num_tiles
        M_nk = M[n,k]; M_nn =M[n,n]
        M[n,n] = @spawn dsyrk(fetch(M_nk),fetch(M_nn))
        for m = n+1:num_tiles
            M_mk = M[n,k]; M_mn = M[m,n]
            M[m,n] = @spawn dgemm(fetch(M_mk),fetch(M_nk),
                                  fetch(M_mn))
        end
    end
end
```

Performance

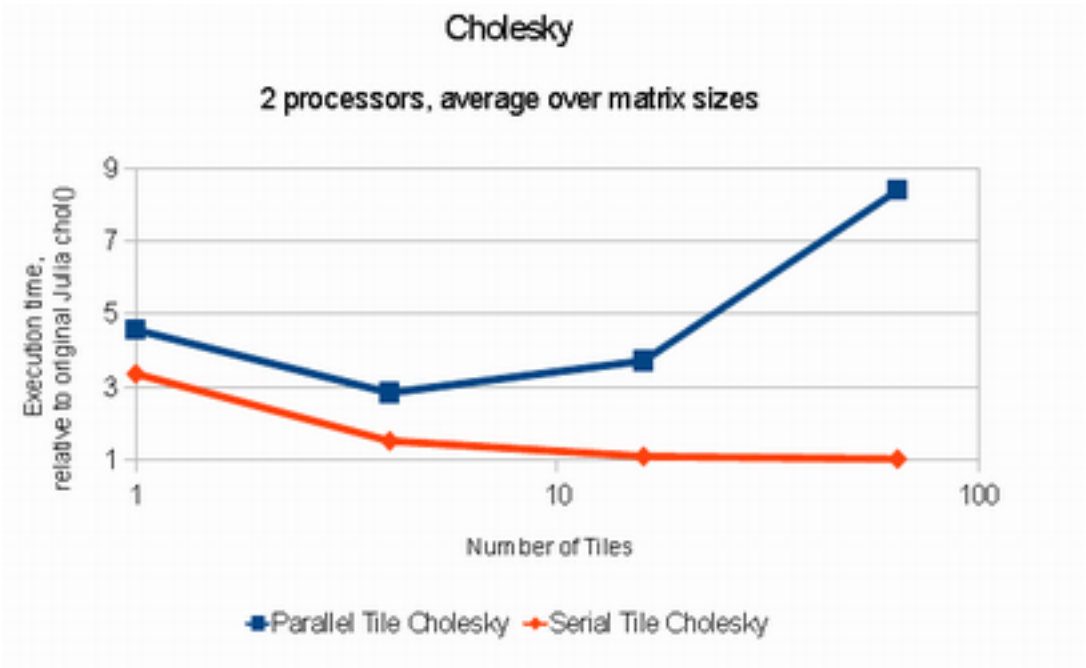
4.1 Practical notes

To test out our code, we ran all tests on the 80-core julia.mit.edu machine with randomly generated matrices (made symmetric for Cholesky with B = A*A'). Since the openBLAS library already supports basic, automatic multithreading, we limited the number of processors in this case to 4 (though more testing could probably find a more optimal solution). Further, since Julia's just-in-time compiler will execute code more quickly on subsequent runs, so we run every test twice and take the elapsed time of the second trial only. Our code can be found in a forked Julia repository on GitHub under the username intirb, including the testing protocols.

We ran tests with varying matrix size (1000x1000, 2000x2000, 4000x4000, 8000x8000, 16000x16000), number of tiles (1, 4, 16, 64), and number of processors (1, 2, 4, 8). Full tables of the results will be sent with this report, but some relevant findings are presented and discussed below. Ultimately, these tests are good ballpark figures but should probably be considered incomplete.

4.2 Relationship between block-size and speed

Intuitively, the speed of a tile-based matrix algorithm should depend on the total number of tiles. The nature of this relationship, however, is complex. A higher number of tiles for a given matrix means that more tile-based operations need to be executed, but each tile-based operation should run faster since the individual tiles are smaller. For the parallel code, more tiles means more opportunity to run operations in parallel but also more need for values to be passed between nodes. For two processors, the results for both cholesky and QR can be found below:



These results illustrate a couple of interesting points. First, while the tiled Cholesky code consistently runs slower than the original Cholesky function implemented in Julia, the tiled

QR code consistently runs faster. Second, in both algorithms, the serial versions seemed to achieve slight gains in speed for an increasing number of tiles (at least in the range tested), while the relationship between number of tiles and speed is somewhat more complex in the parallel algorithms.

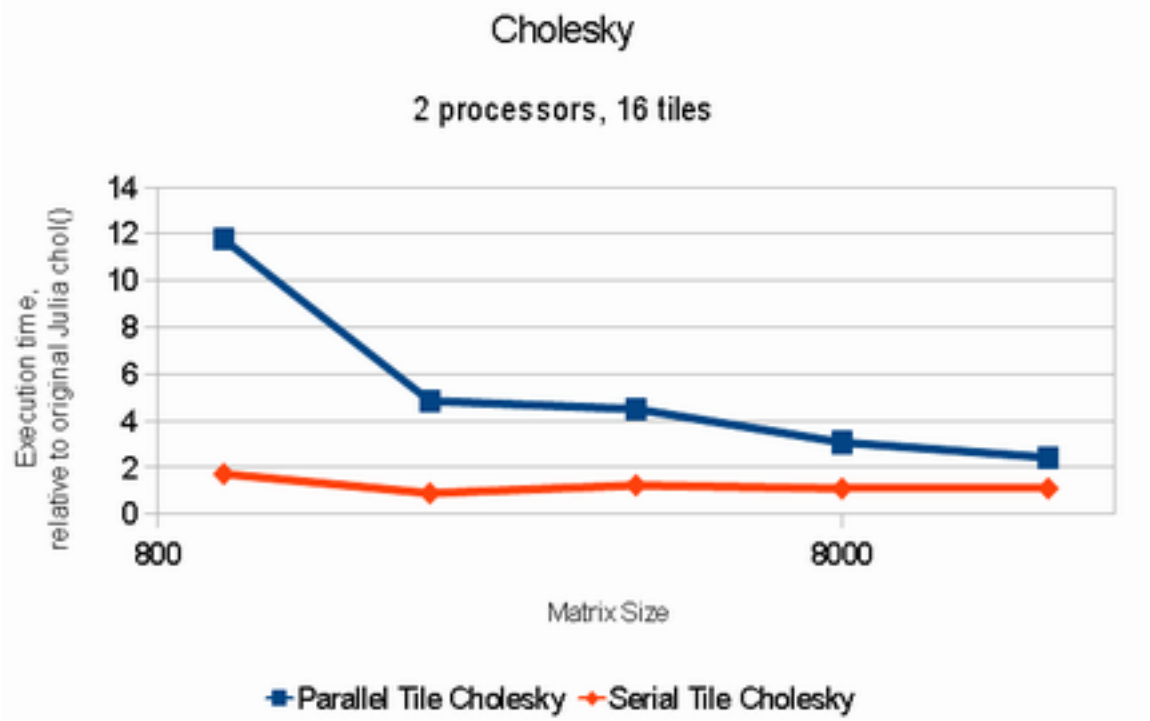
In Cholesky, the serial tiled algorithm approaches the performance of the original as the number of tiles increases, whereas the parallel algorithm seems to achieve its maximum performance at 4 tiles (except in the largest 1600x1600, where 16 tiles was the best). From observation, these trends were very consistent across all matrix sizes. Though it is possible that the increase in performance for the Cholesky code is simply due to the Julia compiler "warming up", the fact that these trends are consistent across matrix sizes and multiple trials suggests that these gains are real. The results for the parallel case demonstrate the limits of parallelization: while increasing the number of tiles improved performance initially, further increases meant a much larger communication overhead between nodes, which bogs down the calculations.

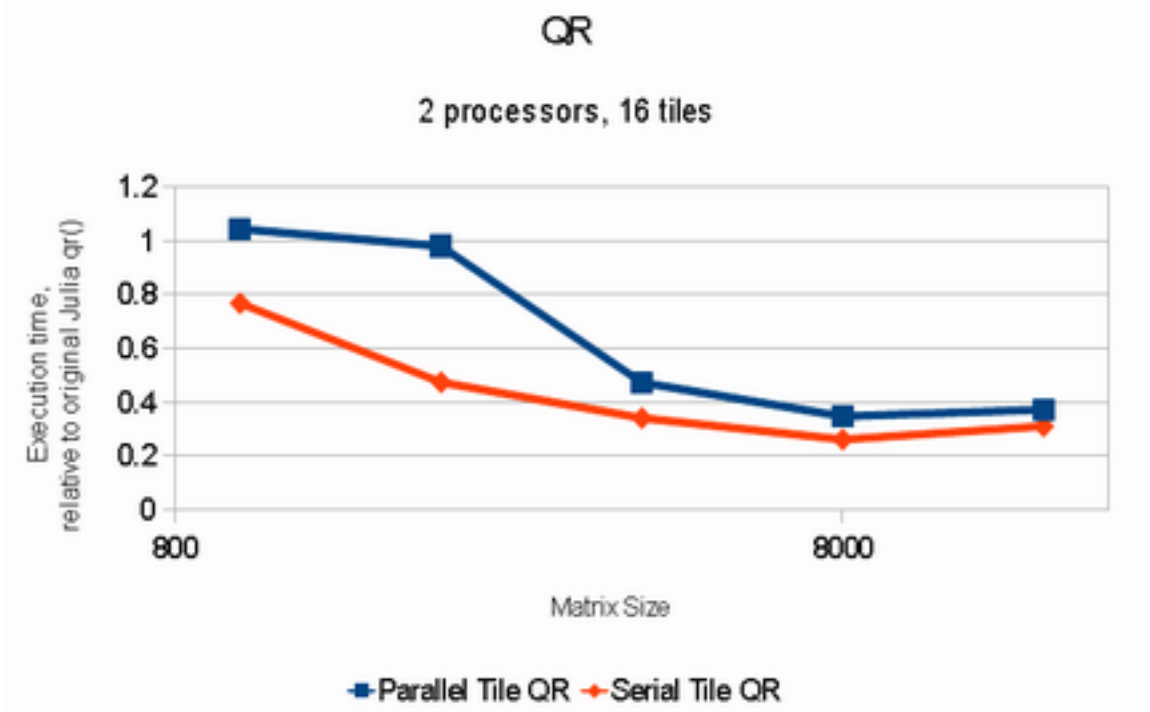
These gains with increasing number of tiles can also be seen in the serial version of QR, though the data here is a little noisier. However, for parallel QR, the optimum choice of tiles is more difficult to determine. The average data seems to suggest that the optimum solution is to forgo tiling completely, but a closer look at the data reveals that the best choice for tile number varies considerably with matrix size, so plotting the averages obscures the results. The results from the parallel QR highlight the difficulty in developing automatic methods for choosing tile size, and likely any heuristic will ultimately sacrifice performance for ease-of-use.

4.3 Number of processors, matrix size, and speed

Given the above results, it can already be guessed that increasing the number of processors is detrimental to speed for the tiled algorithms. In both cases, the parallel tiled algorithms perform slower than their serial counterparts, and adding more processors always slows execution time. Parallel algorithms that perform slower than their serial counterparts can still sometimes be useful, as when one needs to perform an operation on a matrix that is too big for one processor alone.

The issue of speed vs matrix size is important for a number of reasons. If the parallel tile algorithm becomes impractically slow for large matrices, it loses its one advantage (being able to perform operations on large matrices). Similarly, though tile-based Cholesky (QR) performs slower (faster) under the tested conditions, perhaps those trends no longer hold as matrix size gets larger. The results are shown below:





Overall, performance of the tiled algorithms tends to improve relative to existing Julia functions as matrix size gets larger, at least in the range tested. Notably, the gaps in performance between parallel and serial versions also decreases considerably as the matrix size gets larger, suggesting that the parallel algorithms would be appropriate for extremely large matrices when serial code is no longer sufficient.

Future directions

5.1 Other Numerical Algorithms

Currently, we still have difficulty implementing the pseudo code for the LU decomposition described in [1]. Apart from the notation ambiguity, we currently get stuck at the decomposition $A=P_1L_1...P_iL_iU$, unable to do the trick for column-based decomposition to reorganize it into $A=P^*L^*U$ due to the structure of L_i.

More ambitiously, given the building block of Cholesky, QR and LU decomposition, the next important step would be to implement a tile-paralleled version of SVD decomposition, which is widely used in scientific computation.

5.2 Code Optimization and Characterization

The results of performance suggests two potential approaches for optimizing the Cholesky and QR code. While parallelization does seem to improve speeds in Cholesky, both the serial and parallel tiled algorithms are slow relative to previous implementation. Thus, improvements in the way the structure of the code or implementation of the algorithm might be able to increase performance overall. For QR, however, both serial and parallel tiled algorithms run faster relative to previous implementation, though the parallel version never achieves clear gains in performance, suggesting that a more optimized approach to parallelization is needed. Ultimately, testing of both codes was only done under a very limited set of conditions, so of course expanding the conditions of testing would also be helpful to characterize the performance of these functions.

There are a few other ways our code could be optimized or made easier to use. Automatic selection of tiles would help improve usability, though at the expense of performance. Additionally, a few groups are currently working on implementing other features, such as dynamic tile sizes and update order, to improve the performance of the algorithms.

5.3 Code Integration

Right now our code can be found in a forked repository on GitHub. It would be great to eventually integrate all of the BLAS used in our code into the lapack.jl module for calling those functions from the libopenblas shared library.

References

[1] Buttari, A. and Langou, J. and Kurzak, J. and Dongarra, J. A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing 35(1):38--53,2009
[2] OpenBLAS: http://xianyi.github.com/OpenBLAS/
[3] Bosilca, G. and Bouteiller, A. and Danalis, A. and Faverge, M. and Haidar, A. and Herault, T. and Kurzak, J. and Langou, J. and Lemarinier, P. and Ltaief, H. Distributed dense numerical linear algebra algorithms on massively parallel architectures: Dplasma. University of Tennessee Computer Science, Tech. Rep. Technical Report, UT-CS-10-660, 2010
[4] Julia Language: http://julialang.org/
[5] Sharp, John A., editor. Data flow computing: theory and practice. Ablex Publishing Corp, 1992