A performance study on LU decomposition without blocking strategies between C++ and Julia

Nguyen Hoang

Abstract

LU factorization has been known as a good solution to solving linear system of equations for its numerical accuracy and its space complexity over finding inverse of the matrix. Julia is a new on the rise programming language designed for high performance as it "runs like C but reads like Python". So this project will be a study on how well the two languages perform and scale with respect to size and different parallel platform settings. Furthermore, we will also look into a special case of LU factorization, which is the Cholesky decomposition for symmetric positive definite matrices.

1 Introduction

Numerous real life applications that can be modeled will take up the form of solving the following matrix system: $Ax = b$, where A is a known $m \times n$ matrix, b is a known $m \times 1$ vector and x is an unknown $n \times 1$. The simplest solution to this system is to find the inverse of matrix A such that $x = A^{-1}b$. An important aspect of doing numerical computations is the computation cost and calculating the matrix inverse is known to be computationally expensive, especially large matrix, as it is $O(n^3)$.

Then comes the LU decomposition method. LU decomposition factors a matrix as the product of a unit lower triangular matrix and an upper triangular matrix or $A = LU$. It is one of the common methods used to solve square systems of linear equation. Even though the factorization is also $O(n^3)$ but it provides the numerical stability which finding the inverse might not provide and this is especially important where we need to preserve numerical accuracy.

Furthermore, I will also consider the case that A is a Hermitian, symmetric positive-definite matrix then the problem of factorization is a lot simpler when using Cholesky decomposition to get $A = LL^T$ because generalization of the property that a positive real number has a unique positive square root. Even though the runtime of Cholesky algorithm is still $O(n^3)$ but the amortized time complexity is faster than Doolittle's algorithm and we only need to store matrix L. Additionally, the Cholesky algorithm can utilize the either row or column structure to get the most performance speedup for row-major or column-major programming language due to cache reading. For simplicity of this project, I will only consider $n \times n$ matrix A in order to easily scale the size.

2 Methodology

The LU decomposition will be implemented using Doolittle's method shown below and the Cholesky-Crout algorithm will be used for Cholesky factorization. Parallel regions will be created in order to speed up the nested for loops of partial sums and assignments. As we can see below, for Doolittle algorithm, the k-loop is parallelizable with shared memory of A, L and LU while both i-loop and j-loop are parallelizable with a scheduler. While for Cholesky-Crout algorithm, we will use shared memory of A, L and only i-loop is needed to be parallelizable. Both static and dynamic types will be considered for C++. However, according to (1), Julia only considers using static scheduler currently, using all threads and assigning equal iteration counts to each but will be changed in future update to be dynamic.

Algorithm 1 Doolittle's algorithm

Input: $N \times N$ matrix A
Output: $N \times N$ lower triangular matrix L, $N \times N$ upper triangular matrix U

function DOOLITTLE(A, N)
  $L, U \leftarrow zeros(N, N)$
  for $k \leftarrow 1$ to N **do**
    for $i \leftarrow k$ to N **do**
      $U_{k,i} \leftarrow A_{k,i} - \sum_{l=1}^{k-1} L_{k,l}U_{l,i}$
    end for
    for $j \leftarrow k+1$ to N **do**
      $L_{j,k} \leftarrow \frac{A_{j,k} - \sum_{l=1}^{k-1} L_{j,l}U_{l,k}}{U_{k,k}}$
      $L_{j,j} \leftarrow 1$
    end for
  end for
  return L, U
end function

Algorithm 2 Cholesky factorization (column version)

Input: $N \times N$ matrix A
Output: $N \times N$ lower triangular matrix L

function CHOLESKY-CROUT(A, N)
  $L \leftarrow zeros(N, N)$
  for $i \leftarrow 1$ to N **do**
    $L_{i,i} \leftarrow \sqrt{A_{j,j} - \sum_{j=1}^{i} L_{j,i}^2}$
    for $k \leftarrow i+1$ to N **do**
      $L_{i,k} \leftarrow \frac{A_{i,k} - \sum_{l=1}^{i} L_{l,k}L_{l,i}}{L_{i,i}}$
    end for
  end for
  return L
end function

For this project, I will consider the cases of using 1,4,8,16 and 32 threads for running C++ code on both

Cholesky and Doolittle algorithm. On the other hand, for Julia, even though I set the number of Julia threads to be 16 for multi-threading the algorithm, Julia has already used OpenBLAS multi-threading innately. Since the Doolittle's algorithm only has one version so I implement it in both Julia and C++. However, Crout's algorithm can be either in row-major or column-major order so I implement it in row-major order for C++ and in both order for Julia in order to compare the effects of ordering to cache efficiency.

Here is a couple of things to take note of Julia after setting up the programs and doing trial runs of them since the language itself is still relatively new:

- Julia can spend quite some time on precompilling the packages and setting up the package registries on the HPC cluster. The process can take up to 5-10 minutes.
- The first call on any function will take more time than those after that since Julia also need to compile the function itself first.
- Since the language is relatively new, the multi-threading functionality was experimental a couple of changelogs ago but it is still worth figuring out how the scaling works with multi-threading.
- The benchmarking of the algorithms is done by using the package BenchmarkTools.jl, which in order to find the best runtime, it will run the function multiple time where there are multiple samples of runtime, in each sample has multiple function evaluations. So even if the results return as a few seconds for that algorithm, it can actually takes up to several minutes to run in that portion to choose the best runtime from the samples.
- In order to avoid data race in Julia and avoid using atomic addition due to performance issues, I have to create side array to store value then add them up later, this creates tons of work for garbage collector and a lot of memory allocations, which makes the runtime spike up quite a lot.
- Performance of Julia can be improved with a mixed of macros and good Julia techniques. For example, according to (5):
  - @*fastmath* allows floating point optimizations that are correct for real numbers, but lead to differences for IEEE numbers.
  - @*simd* in front of for loops promises that the iterations are independent and may be reordered

3 Results

I have included the raw results and the speedup tables for both algorithms below in the Appendix section. The tables have the columns as the size of the problems, starting from 128 to 8192. The rows of the tables are:

- JLSeq: The sequential version in Julia
- JLSeqRow: The sequential version in Julia in row-major order
- JLSeqRow: The sequential version in Julia in column-major order
- JLFM: The sequential version in Julia with macro extensions to speedup
- JLTh: The threaded version in Julia
- CPPSeq: The sequential version in C++.
- CPPSta*: The parallelized version in C++ with static scheduler where * is the number of threads.
- CPPDym*: The parallelized version in C++ with dynamic scheduler where * is the number of threads.

Firstly, when we compare table 2 with table 4, we can see that the number of threads has much larger impact on larger problem size in Cholesky algorithm than Doolittle algorithm in order to speed up the programs. Since the Doolittle's algorithm reads in the data from the matrix both along the row and column so on either Julia and C++, there will be a lot of cache misses, leading to inefficiency. Furthermore, the speedup from multi-threading in C++ seems to have a cap under 10 times the sequential algorithm while for the Cholesky algorithm, it is more proportional to the number of threads, making it much more efficient to use the threads due to less cache misses. Hence, the ordering of the memory storage plays a big role in the efficiency of the parallelism.

Secondly, when we compare the speed up of Julia in both table 2 and table 4, Julia seems to do a good job for small problem size but then as the problems get big, it fails to reach the standard of C++ for Doolittle's algorithm. However, for Crout's algorithm, even though Julia is using the sequential version, but with the right ordering and with some basic composability of the language (the macro expansions), it can easily outperforms for problem of small to medium sizes and only underperforms 16 and 32 threads dynamic scheduler version of C++.

Finally, I'm highly suspicious of the results of the threaded version of both algorithms. It shows no visible pattern for the proportionality with the size of the problem and at the same time takes a lot of memory allocation as mentioned above. It takes me nearly 2 hours to run the problem up to size 2048 so that is the main reason why I don't let the program run up to size 8196 as it might takes days.

4 Conclusions

In conclusion, from this individual project of mine, I have learned a lesson that when consider parallelism to a problem, the architecture of a programming language is very important to go along with the structure of the algorithm problem. Even a sequential version of the algorithm with a suitable programming language architecture can outperform a lot of parallelized versions of the algorithm with wrong architecture. Julia has shown some potentials, however, it can takes quite some time before it can shine when comparing with big players like C++.

Appendix

	128	256	512	1024	2048	4096	8192
JLSeq	853.803 μs	6.601 ms	218.174 ms	1.885 s	20.504 s		
JLFM	841.921 μs	8.404 ms	283.728 ms	2.520 s	53.260 s		
JLTh	220.653 ms	27.193 s	3.454 s	75.426 s	85.467 s		
CPPSeq	2 ms	18 ms	152 ms	1.226 s	11.856 s	124.834 s	1295.989 s
CPPSta4	2 ms	9 ms	53 ms	378 ms	3.229 s	39.904 s	552.738 s
CPPSta8	1.132 s	10 ms	42 ms	246 ms	30.174 s	72.122 s	456.558 s
CPPSta16	1 ms	4 ms	19 ms	126 ms	1.124 s	27.975 s	252.296 s
CPPSta32	20 ms	45 ms	106 ms	247 ms	1.867 s	35.693 s	311.359 s
CPPDym4	2 ms	9 ms	54 ms	381 ms	3.229 s	38.728 s	416.778 s
CPPDym8	2 ms	10 ms	42 ms	2.089 s	31.629 s	74.813 s	455.262 s
CPPDym16	1 ms	4 ms	19 ms	126 ms	1.120 s	28.206 s	257.156 s
CPPDym32	19 ms	33 ms	78 ms	220 ms	1.878 s	35.503 s	301.027 s

Table 1: Raw results for Doolittle's algorithm

	128	256	512	1024	2048	4096	8192
JLSeq	2.3424607	2.7268596	0.6966916	0.6503979	0.5782286		
JLFM	2.3755198	2.1418372	0.5357244	0.4865079	0.2226061		
JLTh	0.0090640	0.0006619	0.0440069	0.0162543	0.1387202		
CPPSeq	1.0	1.0	1.0	1.0	1.0	1.0	1.0
CPPSta4	1.0000000	2.0000000	2.8679245	3.2433862	3.6717250	3.1283581	2.4792324
CPPSta8	0.0017668	1.8000000	3.6190476	4.9837398	0.3929211	1.7308727	2.8386076
CPPSta16	2.0000000	4.5000000	8.0000000	9.7301587	10.5480427	4.4623414	5.1367798
CPPSta32	0.1000000	0.4000000	1.4339623	4.9635628	6.3502946	3.4974365	4.1623624
CPPDym4	1.0000000	2.0000000	2.8148148	3.2178478	3.6717250	3.2233526	3.1095427
CPPDym8	1.0000000	1.8000000	3.6190476	0.5868837	0.3748459	1.6686137	2.8466883
CPPDym16	2.0000000	4.5000000	8.0000000	9.7301587	10.5857143	4.4257959	5.0396996
CPPDym32	0.1052632	0.5454545	1.9487179	5.5727273	6.3130990	3.5161536	4.3052251

Table 2: Speedups for Doolittle's algorithm

	128	256	512	1024	2048	4096	8192
JLSeqRow	425.150 μs	4.195 ms	145.705 ms	1.216 s	11.717 s		
JLSeqCol	287.251 μs	2.356 ms	20.044 ms	163.861 ms	1.649 s		
JLColFM	93.686 μs	505.401 μs	3.937 ms	29.347 ms	535.131 ms		
JLColTh	47.969 s	155.741 s	464.989 s	15.130 s	107.511 s		
CPPSeq	1 ms	9 ms	75 ms	595 ms	4.79 s	38.797 s	307.483 s
CPPSta4	1 ms	6 ms	39 ms	274 ms	2.097 s	16.736 s	133.605 s
CPPSta8	1 ms	4 ms	24 ms	149 ms	1.131 s	9 s	71.77 s
CPPSta16	52 ms	4 ms	13 ms	85 ms	598 ms	4.638 s	37.453 s
CPPSta32	5 ms	9 ms	33 ms	187 ms	687 ms	3.853 s	26.960 s
CPPDym4	1 ms	6 ms	30 ms	186 ms	1.32 s	10.202 s	79.437 s
CPPDym8	1 ms	5 ms	22 ms	105 ms	681 ms	5.168 s	40.862 s
CPPDym16	1 ms	4 ms	21 ms	74 ms	378 ms	2.784 s	21.213 s
CPPDym32	4 ms	9 ms	24 ms	89 ms	440 ms	3.088 s	22.202 s

Table 3: Raw results for Cholesky-Crout's algorithm

	128	256	512	1024	2048	4096	8192
JLSeqRow	2.35211	2.14541	0.51474	0.48931	0.40881		
JLSeqCol	3.48128	3.82003	3.74177	3.63113	2.90479		
JLColFM	10.67395	17.80764	19.05004	20.27464	8.95108		
JLColTh	0.00002	0.00006	0.00016	0.03933	0.04455		
CPPSeq	1.0	1.0	1.0	1.0	1.0	1.0	1.0
CPPSta4	1.00000	1.50000	1.92308	2.17153	2.28422	2.31818	2.30143
CPPSta8	1.00000	2.25000	3.12500	3.99329	4.23519	4.31078	4.28428
CPPSta16	0.01923	2.25000	5.76923	7.00000	8.01003	8.36503	8.20984
CPPSta32	0.20000	1.00000	2.27273	3.18182	6.97234	10.06930	11.40516
CPPDym4	1.00000	1.50000	2.50000	3.19892	3.62879	3.80288	3.87078
CPPDym8	1.00000	1.80000	3.40909	5.66667	7.03377	7.50716	7.52491
CPPDym16	1.00000	2.25000	3.57143	8.04054	12.67196	13.93570	14.49503
CPPDym32	0.25000	1.00000	3.12500	6.68539	10.88636	12.56380	13.84934

Table 4: Speedups for Cholesky-Crout's algorithm

References

[1] Bezanson, Jeff, et al. "Julia: A Fresh Approach to Numerical Computing." SIAM Review, vol. 59, no. 1, 2017, pp. 65–98., doi:10.1137/141000671.

[2] Wikipedia contributors. "LU decomposition." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 13 May. 2021. Web. 03 Jun. 2021.

[3] Wikipedia contributors. "Cholesky decomposition." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 19 May. 2021. Web. 03 Jun. 2021.

[4] "Multi-Threading." Multi-Threading · The Julia Language, docs.julialang.org/en/v1/manual/multi-threading/.

[5] "Performance Tips." Performance Tips · The Julia Language, docs.julialang.org/en/v1/manual/performance-tips/.

[6] JuliaLanguage, director. Shared Memory Parallelism in Julia with Multi-Threading — Cambridge Julia Meetup (May 2018). YouTube, 17 May 2018, https://www.youtube.com/watch?v=YdiZa0Y3F3c.

[7] A Quick Introduction to Data Parallelism in Julia, juliafolds.github.io/data-parallelism/tutorials/quick-introduction/.