Parallel Cholesky Decomposition in Julia (6.338 Project)

Omar Mysore

December 16, 2011

Project Introduction

Sparse matrices dominate numerical simulation and technical computing. Their applications are practically limitless, ranging from solving partial differential equations to convex optimization to big data. Because of the wide use of sparse matrices, the objective of this project was to implement sparse Cholesky decomposition in parallel using the Julia language. In addition to developing the Cholesky decomposition feature in the Julia language and investigating the effects of parallelization, this project served the purpose of aiding the development of sparse matrix support in Julia.

A working sparse parallel Cholesky decomposition solver was developed. Although, the current implementation contains limits in terms of speed and capability, it is hoped that this implementation can serve as a means to further development. The remainder of this report will discuss the basics of Cholesky decomposition and the algorithm used, the opportunities and methods of parallelization, the results, and the potential for future work.

Cholesky Decomposition

The Cholesky decomposition of a symmetric positive definite matrix A determines the lower-triangular matrix L, where $LL^T = A$. Although it is limited to symmetric positive

definite matrices, these matrices often appear in fields such as convex optimization. The following equations are used from [2].

The basic dense Cholesky decomposition algorithm consists of repeatedly performing the factorization below on a matrix, A of size n, where d is a constant value and d and v is an n-1 by 1 column vector.

$$A = \begin{pmatrix} d & v^t \\ v & C \end{pmatrix} = \begin{pmatrix} \sqrt{d} & 0 \\ v/\sqrt{d} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - vv^t/d \end{pmatrix} \begin{pmatrix} \sqrt{d} & v^t/\sqrt{d} \\ 0 & I \end{pmatrix}$$

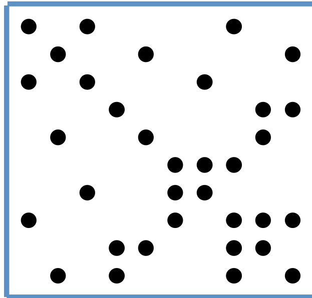
Once this factorization is completed, the first column of L is determined. Then the same factorization is performed on C-vv/d, and the process is repeated until all of the columns of L are found. Similarly, the same process can be done for block matrices:

$$A = \begin{pmatrix} B & V^t \\ V & C \end{pmatrix} = \begin{pmatrix} L_B & 0 \\ VL_B^{-t} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - VB^{-1}V^t \end{pmatrix} \begin{pmatrix} L_B^t & L_B^{-1}V^t \\ 0 & I \end{pmatrix}$$

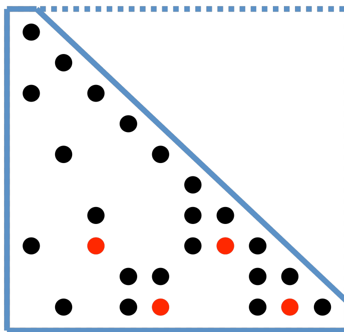
Cholesky Decomposition for Sparse Matrices

For sparse matrices, several additional steps are taken in order to take advantage of the substantially fewer nonzero values. First the fill-ins, and the structure of L are determine, without calculating the values. Next the tree of dependencies is found, and finally the values of L are calculated. For a detailed explanation of the method summarized in this report, see [2].

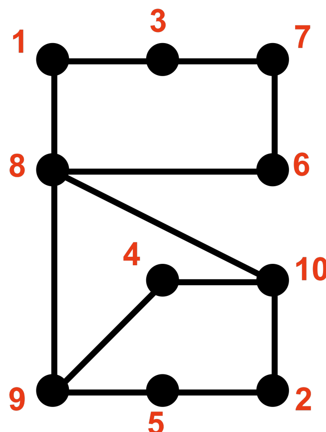
The first step in sparse Cholesky decomposition is to determine the structure of L without explitly determining L. All of the following images are used from John Gilbert's slides [1]. Suppose A has the structure below:



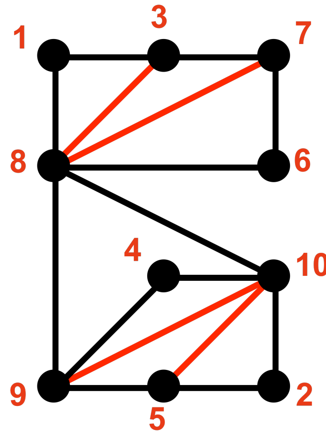
Then, the goal would be to determine the structure of L, which is show below:



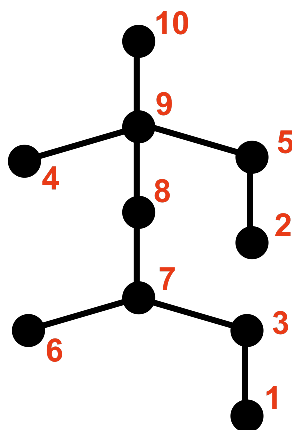
The red dots are known as the fill-ins, since these values are nonzero in L, but zero in A. Although the structure of L is known, none of the specific values are known. In order to determine the structure of L, the graphs of the matrices are used. Below is the graph of the previously shown matrix A:



From this graph, the graph of L can easily be determined by connecting the higher numbered neighbors of each node/column in the graph. Below is the graph of L with the fill-ins in red:



Once the structure of L is determined, the dependency tree can be determined. In order to calculate the dependency tree, the parent of each node must be determined, and the parent of a given node/column is the minimum row value of a nonzero value in the given column of L, not including the diagonal values. For example, for the graph of L previously shown, the dependency tree is shown below:



After the dependency tree is determined, the values of L can be calculated. The basic equations for determining each column of L are shown below:

For each column j in A

Determine the frontal matrix Fj, where

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i_1} & \cdots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{pmatrix} + \overline{U}_j$$

And,

$$\overline{U}_j = - \sum_{k \in T[j]-\{j\}} \begin{pmatrix} \ell_{j,k} \\ \ell_{i_1,k} \\ \vdots \\ \ell_{i_r,k} \end{pmatrix} \begin{pmatrix} \ell_{j,k} & \ell_{i_1,k} & \cdots & \ell_{i_r,k} \end{pmatrix}$$

Here, T[j]-{j} represent all of children nodes, which have to already have been determined in order to calculate Lj.

This is the basic formulation for determining the sparse Cholesky decomposition. All equations and images used in this and the previous section were obtained from [1] and [2]. For more details see [2].

Parallelization and Implementation in Julia

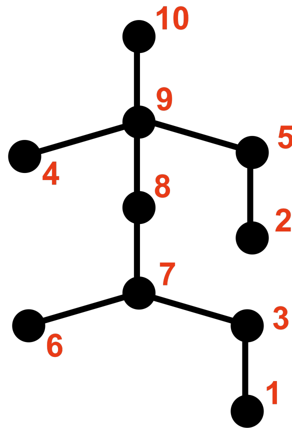
For sparse Cholesky decomposition, there are three primary opportunities for parallelization. First, in determining the structure of L. Second, in simultaneously computing columns, which are independent. Third, in turning the addition step for each column into a parallel reduction.

In order to determine the structure of L, each column or block of columns can be sent to different processors, and the necessary fill-ins can be determined. Once the fill-ins are determined, they can be added to L. The following Julia function demonstrates this:

```
function fillinz(L)
 L=tril(L)
 L=spones(L)
 refs=[remote_call((mod(i,nprocs()))+1,pfillz,L[:,i],i)|i=1:size(L)[1]
 for i=1:size(L)[1]
   q=fetch(refs[i])
   for j=1:length(q)/2
     L[q[2*j-1],q[2*j]]=1
   end
 end
return L
end
```

The input to this function is the matrix A, for which we would like the Cholesky decomposition. The vector, refs, sends each column to different processors, and the for-loop obtains the results and adds the fill-ins.

The tree structure of dependencies allows for further parallelization. As previously discussed, for a matrix A, the dependency tree might look like the following figure:



In this case, columns 1,2,4, and 6 of L can all be determined without any other columns of L, and they can be determined simultaneously. The following for-loop, which is part of the main sparse Cholesky function, performs this process of going through the levels of the tree and sending all of the columns at the same level to different processors:

```
for i=1:size(kids)[1]
    k=[]
    for j=1:size(kids)[1]
      if tree[j]==i-1
        k=vcat(k,[j])
      end
     end
      refs=[remote_call(mod(i,nprocs())+1,spcholhelp,A,L,k[i],kids)|i=1:length(k)]
      for m=1:length(refs)
        lcol=fetch(refs[m])
        L[:,k[m]]=lcol
      end
end
```

In this for loop, the index i loops through all of the possible values for the number of children a column can have. The vector refs contains all of the columns of L that are calculated in parallel for a given stage of the tree.

The final level of parallelization is within the function which determines the values of the columns L. During the calculation, a number of matrices equal to the number of

children of the given column must be added. Rather than adding serially, this is done with a parallel for-loop. It is shown below:

```
addz = @parallel (+) for i=1:size(kids)[1]
     -L[nzs,convert(Int16,kids[i])]*L[nzs,convert(Int16,kids[i])]'
  end
```

Results and Discussion

The primary objective of the project was to write a function, which performs Cholesky decomposition on a sparse matrix. This function is called spchol(), and the input argument is the matrix. This function seems to work for all matrices tested. A very simple example is shown:

```
julia> A
10x10 Float64 Array:
 121.0  33.0  0.0  0.0  33.0    0.0  55.0   0.0  44.0   0.0
  33.0  10.0  0.0  0.0  12.0    0.0  17.0   2.0  12.0   0.0
   0.0   0.0  1.0  0.0   0.0    9.0   0.0   0.0   0.0   0.0
   0.0   0.0  0.0  1.0   8.0    0.0   0.0   0.0   0.0   0.0
  33.0  12.0  0.0  8.0  83.0    0.0  23.0   6.0  12.0   2.0
   0.0   0.0  9.0  0.0   0.0  162.0  18.0  63.0   0.0   0.0
  55.0  17.0  0.0  0.0  23.0   18.0  38.0  18.0  20.0   4.0
   0.0   2.0  0.0  0.0   6.0   63.0  18.0  54.0   0.0   3.0
  44.0  12.0  0.0  0.0  12.0    0.0  20.0   0.0  17.0   0.0
   0.0   0.0  0.0  0.0   2.0    0.0   4.0   3.0   0.0  14.0

julia> u=spchol(A)
10x10 Float64 Array:
 11.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
  3.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
  3.0  3.0  0.0  8.0  1.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  9.0  0.0  0.0  9.0  0.0  0.0  0.0  0.0
  5.0  2.0  0.0  0.0  2.0  2.0  1.0  0.0  0.0  0.0
  0.0  2.0  0.0  0.0  0.0  7.0  0.0  1.0  0.0  0.0
  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
  0.0  0.0  0.0  0.0  2.0  0.0  0.0  3.0  0.0  1.0

julia> u*u'
10x10 Float64 Array:
```

```
 121.0  33.0  0.0  0.0  33.0    0.0  55.0   0.0  44.0   0.0
  33.0  10.0  0.0  0.0  12.0    0.0  17.0   2.0  12.0   0.0
   0.0   0.0  1.0  0.0   0.0    9.0   0.0   0.0   0.0   0.0
   0.0   0.0  0.0  1.0   8.0    0.0   0.0   0.0   0.0   0.0
  33.0  12.0  0.0  8.0  83.0    0.0  23.0   6.0  12.0   2.0
   0.0   0.0  9.0  0.0   0.0  162.0  18.0  63.0   0.0   0.0
  55.0  17.0  0.0  0.0  23.0   18.0  38.0  18.0  20.0   4.0
   0.0   2.0  0.0  0.0   6.0   63.0  18.0  54.0   0.0   3.0
  44.0  12.0  0.0  0.0  12.0    0.0  20.0   0.0  17.0   0.0
   0.0   0.0  0.0  0.0   2.0    0.0   4.0   3.0   0.0  14.0

julia> A
10x10 Float64 Array:
 121.0  33.0  0.0  0.0  33.0    0.0  55.0   0.0  44.0   0.0
  33.0  10.0  0.0  0.0  12.0    0.0  17.0   2.0  12.0   0.0
   0.0   0.0  1.0  0.0   0.0    9.0   0.0   0.0   0.0   0.0
   0.0   0.0  0.0  1.0   8.0    0.0   0.0   0.0   0.0   0.0
  33.0  12.0  0.0  8.0  83.0    0.0  23.0   6.0  12.0   2.0
   0.0   0.0  9.0  0.0   0.0  162.0  18.0  63.0   0.0   0.0
  55.0  17.0  0.0  0.0  23.0   18.0  38.0  18.0  20.0   4.0
   0.0   2.0  0.0  0.0   6.0   63.0  18.0  54.0   0.0   3.0
  44.0  12.0  0.0  0.0  12.0    0.0  20.0   0.0  17.0   0.0
   0.0   0.0  0.0  0.0   2.0    0.0   4.0   3.0   0.0  14.0
```

Above, A and u*u’ are identical.

Tests were conducted to see the effects of parallelization. All matrices were generated in the same manner, by first declaring A=tril(round(10*sprand(n,n,.3))+eye(n)); and then A=A*A’. The results are shown in the table below

Size	Time on 1 processor	Time on 4 processors	Parallel to serial ratio
10x10	0.0025 s	.189 s	76
50x50	0.101 s	.52 s	5
100x100	2.8 s	2.8 s	1
150x150	24.5 s	22.1 s	0.9

For smaller matrices, it appears as though serial is better, because the communication between processors dominates. As the size of matrices increases, parallel seems to substantially improve relative to serial. More tests need to be conducted in order to understand the behavior of this function.

Limitations and Further Work

As stated previously, more tests need to be conducted in order to understand the performance and limitations of the spchol() function. Additionally, a major limitation is the fact that although the algorithm is designed for sparse matrices, it currently only works for full sparse matrices (and of course full matrices). This is also a possible cause of the major time increase with respect to matrix dimensions shown in the results. Initially the algorithm was developed this way, because of errors involving indexing columns of matrics that were declared as sparse. While these errors were recently fixed, the algorithm currently still does not work for sparse matrices unless they are declared as full. Next steps involve debugging this.

Thoughts and Questions About Julia

As stated previously, currently the spchol() function presented in this report, treats all of the matrices and vectors as full. For example, the function tril() is called by spchol():

```
function tril(B)
c=zeros(size(B)[1],size(B)[1])
for i=1:size(B)[1]
 c[i:size(B)[1],i]=B[i:size(B)[1],i]
end
return c
end
```

Clearly this function treats B and c as dense matrices. I believe this raises several questions. Firstly, if sparse matrix functions are implemented in Julia, should they be designed to work for both dense and sparse matrices? For example, if tril assumed B was sparse and accessed B.nzval, then it could never work for dense matrices. Another question this raises is what would cause a function designed for dense matrices not work for a sparse matrix? If I run the spchol() function on a sparse matrix that is declared as a sparse matrix, I get incorrect results or errors. It only works if I make the sparse matrix full. This is a bit problematic, and should be addressed in the future. While attempting to debug this, I found an interesting result:

```
 julia> A
10-by-10 sparse matrix with 21 nonzeros:
        [1,     1]      = 6.0
        [2,     1]      = 9.0
        [3,     1]      = 11.0
        [7,     1]      = 1.0
        [9,     1]      = 2.0
        [2,     2]      = 1.0
        [6,     2]      = 15.0
        [3,     3]      = 1.0
        [6,     3]      = 6.0
        [10,    3]      = 5.0
        [4,     4]      = 1.0
        [8,     4]      = 3.0
        [5,     5]      = 1.0
        [6,     5]      = 6.0
```

```
[6,     6]    = 9.0
[7,     7]    = 1.0
[8,     7]    = 12.0
[8,     8]    = 3.0
[9,     8]    = 1.0
[9,     9]    = 9.0
[10,    10]   = 1.0
```

```
julia> full(A*A')
10x10 Float64 Array:
 36.0  54.0   66.0  0.0  0.0   0.0  6.0  0.0  12.0  0.0
 54.0  82.0   99.0  0.0  0.0  15.0  9.0  0.0  18.0  0.0
 66.0  99.0  122.0  0.0  0.0   6.0  0.0  0.0   0.0  0.0
  0.0   0.0    0.0  1.0  0.0   0.0  0.0  0.0   0.0  0.0
  0.0   0.0    0.0  0.0  1.0   0.0  0.0  0.0   0.0  0.0
  0.0  15.0    0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0
  6.0   0.0    0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0
  0.0   0.0    0.0  3.0  0.0   0.0  0.0  0.0   0.0  0.0
 12.0   0.0    0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0
  0.0   0.0    5.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0

julia> full(A)*full(A')
10x10 Float64 Array:
 36.0  54.0   66.0  0.0  0.0    0.0   6.0    0.0  12.0   0.0
 54.0  82.0   99.0  0.0  0.0   15.0   9.0    0.0  18.0   0.0
 66.0  99.0  122.0  0.0  0.0    6.0  11.0    0.0  22.0   5.0
  0.0   0.0    0.0  1.0  0.0    0.0   0.0    3.0   0.0   0.0
  0.0   0.0    0.0  0.0  1.0    6.0   0.0    0.0   0.0   0.0
  0.0  15.0    6.0  0.0  6.0  378.0   0.0    0.0   0.0  30.0
  6.0   9.0   11.0  0.0  0.0    0.0   2.0   12.0   2.0   0.0
  0.0   0.0    0.0  3.0  0.0    0.0  12.0  162.0   3.0   0.0
 12.0  18.0   22.0  0.0  0.0    0.0   2.0    3.0  86.0   0.0
  0.0   0.0    5.0  0.0  0.0   30.0   0.0    0.0   0.0  26.0
```

Upon inspection, full(A*A’) is not equal to (full(A))*(full(A))’. I suspect that these should be equal, but something seems to be causing a problem.

Conclusion

The function spchol() is presented in this report. As stated in the previous section, some work and testing still needs to be done. I would be happy to contribute in any way possible.

References

[1] John Gilbert's slides from Day 1 of Sparse Matrix Days at MIT. Available at http://www.cs.ucsb.edu/~gilbert/talks/talks.htm.

[2] Liu, Joseph W. H. "The Multifrontal Method for Sparse Matrix Soluation: Theory and Practice. SIAM Review. Vol. 34, No. 1 (Mar., 1992, pp.82-109.

Acknowledgements

I would like to thank Alan Edelman, Jeff Bezanson, and Viral Shah. Additionally I would like to thank everyone who has been developing the Julia language.

Appendix A: Running the code

All of code is found in spcholp.j and must be run with @everywhere load("spcholp.j"). To find the Cholesky decomposition of A, run spchol(A).

Appendix B: The Code for spcholp.j:

```
function spones(A)
 A=sparse(A)
 A.nzval=ones(length(A.nzval),)
 return full(A)
end

function tril(B)
c=zeros(size(B)[1],size(B)[1])
for i=1:size(B)[1]
 c[i:size(B)[1],i]=B[i:size(B)[1],i]
end
return c
end


@everywhere function pfillz(b,i)
   r=[]
   for j=1:(length(b))
      if b[j]==1
       for k=1:(length(b))
         if (j>i) && (k>j) && b[k]==1
         r=vcat(r,[k,j])
         end
       end
      end
    end
    return r
end
```

```
function fillinz(L)
 L=tril(L)
 L=spones(L)

  refs=[remote_call((mod(i,nprocs()))+1,pfillz,L[:,i],i)|i=1:size(L)[1]]

 for i=1:size(L)[1]
   q=fetch(refs[i])
   for j=1:length(q)/2
       L[q[2*j-1],q[2*j]]=1
    end
 end

 return L
 end



function nzindex(g)
b=[]
 for i=1:length(g)
   if g[i]!=0
   b=vcat(b,[i])
 end
end
return b
end

function pary(L)
 u=size(L)
 par=zeros(u[1]-1,1)
 for m=1:(u[1]-1)
   dad=nzindex(L[:,m])
   if size(dad)[1]==1
      par[m]=length(L[:,m])
   else
      par[m]=dad[2]
   end
end
return par
end
```

```
function kiddies(par)

 dim=length(par)+1
 kids=zeros(dim,dim)

 for i=1:length(par)
   kids[i,par[i]]=i
 end

 for j=1:size(kids)[1]
   p=nzindex(kids[:,j])
   for k=1:length(p)
     kids[:,j]=kids[:,j]+kids[:,p[k]]
   end
 end

return kids
end
```

```
@everywhere function frontconstruct(A,L,colj,kids)
 col=L[:,colj]
 nzs=nzindex(col)
 Uj=zeros(length(nzs),length(nzs))
 if size(kids)[1]!=0
   addz = @parallel (+) for i=1:size(kids)[1]

       -L[nzs,convert(Int16,kids[i])]*L[nzs,convert(Int16,kids[i])]'

    end
 else
   addz=zeros(length(nzs),length(nzs))
end

Fj=zeros(length(nzs),length(nzs))

Fj[:,1]=A[nzs,colj]
Fj[1,:]=A[colj,nzs]

F=Fj+Uj+addz

alpha=sqrt(F[1,1]);
r=F[:,1]
lz=vcat(alpha,(1/alpha)*r[2:length(r)])

return lz
end
```

```
@everywhere function spcholhelp(A,L,i,kids)
 kidset=nonzeros(kids[:,i])
 if length(kidset)==0
   kidset=[]
 end
 lz=frontconstruct(A,L,i,kidset)
 for j=1:size(A)[1]
  if L[j,i]==1
    L[j,i]=lz[1]
    lz=lz[2:length(lz)]
   end
  end
 return L[:,i]
end
```

```
function spchol(A)
 B=A
 L=fillinz(B)
 par=pary(L)
 kids=kiddies(par)
 tree=zeros(size(kids)[1])
 for i=1:size(kids)[1]
    tree[i]=length(nonzeros(kids[:,i]))
 end

 for i=1:size(kids)[1]
    k=[]
    for j=1:size(kids)[1]
      if tree[j]==i-1
         k=vcat(k,[j])
       end
     end

      refs=[remote_call(mod(i,nprocs())+1,spcholhelp,A,L,k[i],kids)|i=1:length(k)]

      for m=1:length(refs)
         lcol=fetch(refs[m])
         L[:,k[m]]=lcol
      end
  end


return L

end
```