International Journal of Innovative Research in Engineering & Management (IJIREM)
ISSN: 2350-0557, Volume-3, Issue-6, November-2016
DOI: 10.21276/ijirem.2016.3.6.10

FACTORIZATION, MULTIPLICATION AND VIEWS ON PARALLEL FOR SPARE MATRICES

D.A.GISMALLA
Faculty of Mathematics & Computer Sciences , Gezira University
Wad Medani,P.O.Box 20 , SUDAN

MOHAMED H. A. ELHEBIR
Faculty of Mathematics & Computer Sciences , Gezira University
Wad Medani,P.O.Box 20 , SUDAN

ABSTRACT

The complexity of multiplying two matrices with the natural algorithm that of order O(n^3) has been improved to generate Stassen's algorithm to improve the complexity to be of order O($n^{\ln(7)_2}$) $\approx O(n^{2.8})$ using recursive procedure for divide-and –conquer algorithm[1]. Multiplication of matrices is essential when solving a large system of equation using iterative techniques or direct factorization method. For iterative technique the conjugate-gradient method is viewed [3]. For direct factorization of matrices, the Gram-Schmidt's with complexity (2 mn^2) and Householder's algorithms with complexity (2 m$n^2 - \frac{2}{3}n^3$) , the MATLAB PROGRAM are given. When m and n are large it can be seen that the flops for the three algorithm Stassen , Schmidt and Householder are not differ very much while the iterative techniques specially conjugate-gradient method may be executed in a less time than the others as in [4]. If one looks deeper in these algorithms , all apply the operations to multiply two matrices. So if We reduce the number of flops for multiplication ,the efficiency for times can be achieved . To achieved such goal either to write PAREALLEL algorithms for sparse matrices or investigate LIBARAY ROUTINES IN DIFFERENT LANGUAGES
But the problem with parallel computing to get more efficient programs , all must be run on SUPPER or PARALLEL computing machine which is not available to us to run problems with intensive data ,never the less beginners to parallel computing can grab a little lot of information.
Here , We first , give both the Schmidt's and Householder's algorithms with Matlab Program for them. In these algorithms , all apply the operations to multiply two matrices. Second ,factorization and linear system of equations are considered with some analysis considering Cholesky factors the matrix A=LL^T where L is a lower triangular matrix and the system can be solved with the forward and backward algorithm while the inverse for an upper triangular matrix is given in Fig.(3)for the file INV_LOWER.m
Third, sparse matrices representation
as Compressed Sparse Column format CSC and Compressed Sparse Row format CSR will be given with an algorithm to multiply two matrices as in Fig.(6.).
Fourth, parallelism and multination of matrices algorithm such as FOX'S algorithm and CUDA program are given in Fig.(10) &Fig.(11), respectively. Multiple approaches to demonstrate Parallelism on matrices and solve a system of equation are given in section IV part C . Finally computational remarks are given .

Keywords: Stassen's,schmid's,Householder's Fox's algorithm, CSR,CSC ,CUDA &Sparse

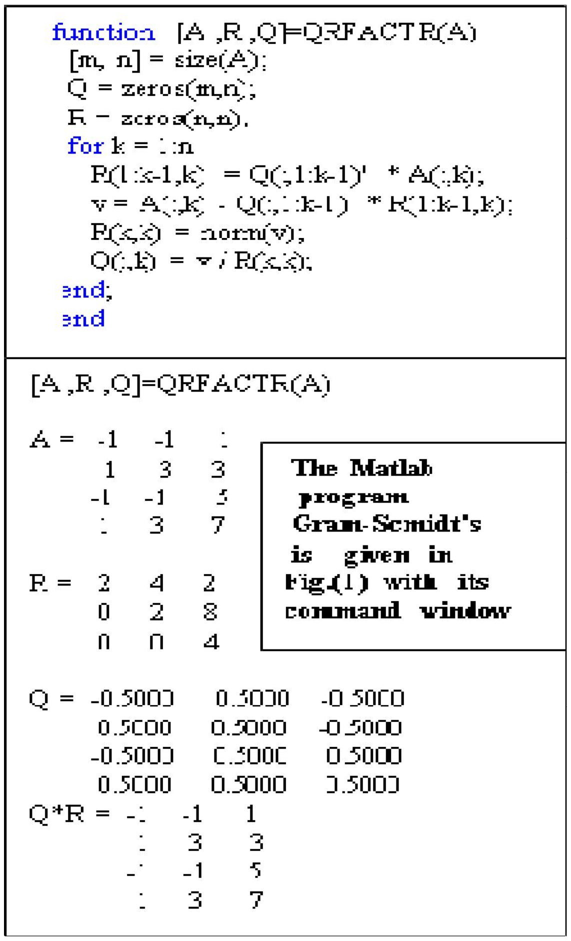
1. ALGORITHMS FOR FACTORIZATIONS

1.1 Gram-Schmidt's Algorithm

The algorithm and its Matlab program

```
Given: m x n matrix A with linearly independent columns $a_1, . . . , a_n$
for k = 1 to n
    $R_{1,k} = q_1{}^T a_k$
    $R_{2,k} = q_1{}^T a_k$
        .. .
        ….
    $R_{k-1,k} = q^T_{k-1} a_k$
$\tilde{q}_k = a_k - (R_{1,k}q_1 + R_{2,k}q_2 + ….. + R_{k-1,k}q_{k-1})$
$R_{k,,k} = \|\tilde{q}_{k,k}\|$
        $q_k = \frac{1}{R_{k,k}}\tilde{q}_k$
  End
```

The Matlab Program Gram-Schmidt's
The Matlab program is given in Fig.(1) withits command window that gives A=QR



```
function  [A ,R ,Q]=QRFACTR(A)
  [m, n] = size(A);
  Q = zeros(m,n);
  R = zeros(n,n);
  for k = 1:n
    R(1:k-1,k)  = Q(:,1:k-1)'  * A(:,k);
    v = A(:,k) - Q(:,1:k-1)  * R(1:k-1,k);
    R(k,k)  = norm(v);
    Q(:,k)  = v / R(k,k);
  end;
 end
```

```
[A ,R ,Q]=QRFACTR(A)

A =   -1    -1     1
       1     3     3
      -1    -1     5
       1     3     7

R =    2     4     2
       0     2     8
       0     0     4

Q =  -0.5000     0.5000    -0.5000
      0.5000     0.5000    -0.5000
     -0.5000     0.5000     0.5000
      0.5000     0.5000     0.5000
Q*R =  -1    -1     1
        1     3     3
       -1    -1     5
        1     3     7
```

The Matlab program Gram-Scmidt's is given in Fig.(1) with its command window

B. Householder's Algorithm

The algorithm for Householder

Householder algorithm overwrites A=$\begin{bmatrix} R \\ 0 \end{bmatrix}$

For k=1 to n

1. Define $y = A_{k:m,k}$ and compute (m-k+1) vector v_k:

$w = y + sign(\ y_1)\|y\|\ e_1\ ,\ v_k = \frac{1}{\|w\|}w$

2. Multiply $A_{k:m,k:n}$ with the reflector $(I - 2v_k{v_k}^T)$:

$$A_{k:m,k:n} = A_{k:m,k:n} - \ 2\ v_k({v_k}^T A_{k:m,k:n})$$

Matlab program Householder algorithm

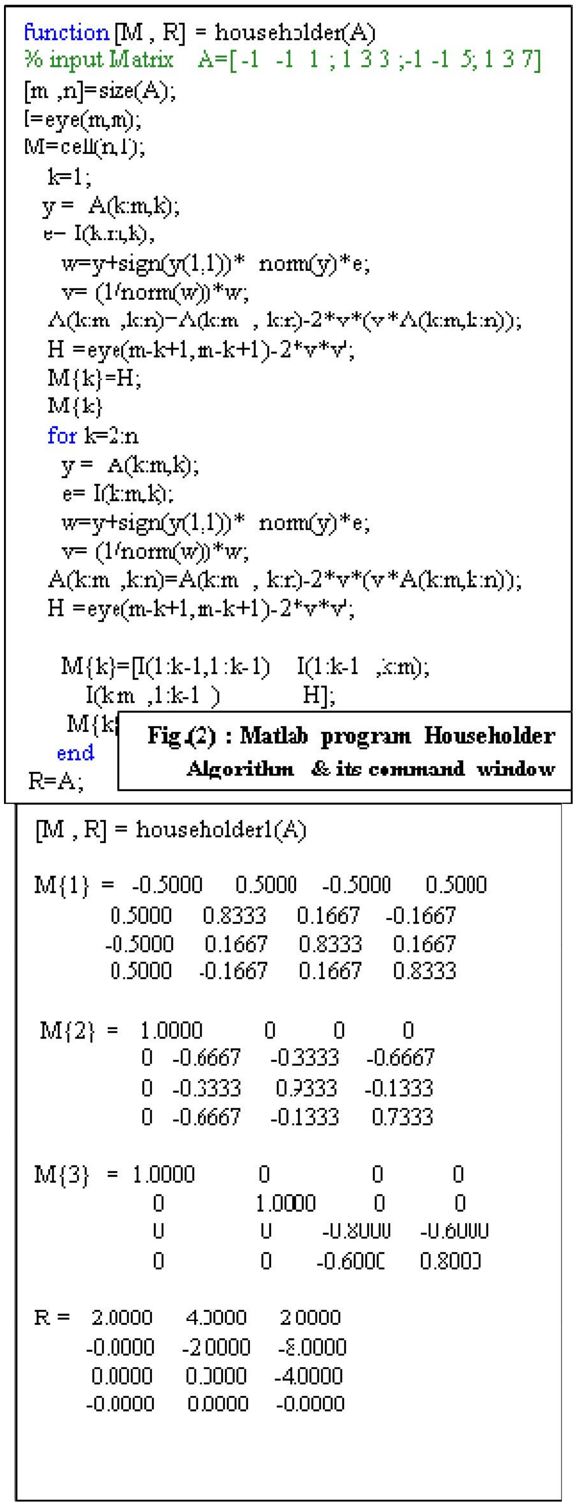
Its Matlab program is in Fig.(2) with the upper triangular matrix R and its orthogonal matrices M{j}=H_j ;j=1 ,2 ,3

Such that M{3} M{2} M{1}R=A (1)

2. FACTORIZATION AND LINEAR SYSTEM OF EUATIONS

2.1. Cholesky algorithm

There are many algorithms applied for factorization of matrices the preferred one is Cholesky when the matrix is symmetric andpositive definite that can bebe found elsewhere



```
function [M , R] = householder(A)
% input Matrix   A=[-1  -1  1 ; 1 3 3 ;-1 -1 5; 1 3 7]
[m ,n]=size(A);
I=eye(m,m);
M=cell(n,1);
  k=1;
 y =  A(k:m,k);
 e= I(k:m,k);
   w=y+sign(y(1,1))*  norm(y)*e;
   v= (1/norm(w))*w;
  A(k:m ,k:n)=A(k:m  , k:n)-2*v*(v*A(k:m,k:n));
  H =eye(m-k+1,m-k+1)-2*v*v';
  M{k}=H;
  M{k}
  for k=2:n
   y =  A(k:m,k);
   e= I(k:m,k);
   w=y+sign(y(1,1))*  norm(y)*e;
   v= (1/norm(w))*w;
  A(k:m ,k:n)=A(k:m  , k:n)-2*v*(v*A(k:m,k:n));
  H =eye(m-k+1,m-k+1)-2*v*v';

   M{k}=[I(1:k-1,1:k-1)   I(1:k-1  ,k:m);
      I(k:m  ,1:k-1 )          H];
   M{k
  end
R=A;
```

Fig.(2) : Matlab program Householder Algorithm & its command window

```
[M , R] = householder1(A)

M{1} =  -0.5000    0.5000   -0.5000    0.5000
         0.5000    0.8333    0.1667   -0.1667
        -0.5000    0.1667    0.8333    0.1667
         0.5000   -0.1667    0.1667    0.8333


M{2} =   1.0000         0         0         0
              0   -0.6667   -0.3333   -0.6667
              0   -0.3333    0.9333   -0.1333
              0   -0.6667   -0.1333    0.7333


M{3}  =  1.0000         0         0         0
              0    1.0000         0         0
              0         0   -0.8000   -0.6000
              0         0   -0.6000    0.8000


R =    2.0000    4.0000    2.0000
      -0.0000   -2.0000   -8.0000
       0.0000    0.0000   -4.0000
      -0.0000    0.0000   -0.0000
```

Cholesky factors the matrix A=LL^T where L is a lower triangular matrix and the system can be solved with the forward and backward algorithm. But we are interested here to pointto the reader that when finding the inverse of the matrix

One get $A^{-1} = {L^T}^{-1} L^{-1}$,then we require to find the inverse of an upper triangular matrix as well for lower triangular matrix. The program in Fiq.(3) , specially INV_LOWER.m, is given to overcome this.

2.2.Gram-Schmidt's algorithm

The Gram-Schmidt's algorithm solve Ax=y with the factorization of A=QR
R is an upper triangular matrices and Q is an orthogonal matrices ,i.e QQ^T=I , the identity matrix. However , the algorithm is sensitivity about accumulations of rounding error . To see , factor A= USV , where U and V
are orthogonal matrices and S is
diagonal$S_{jj} = 10^{-10(j-1)/(n-1)}$,$j = 1,2,\ldots n$

When A is a square matrix of size n=50 , then the orthogonality of Q such QQ^T=I will be lost and the reader must look for another algorithm called the modified Gram-Schmidt's algorithm .However the advantage of this algorithm it can be used to find the eigenvectors for the matrices.

2.3. Householder algorithm

Householder algorithmis used as apreferred method in Matalb language for it is less sensitive about error . The Householder algorithm factor the Matrix A as

$$M\{n\}M\{n-1\}\ldots.M\{1\}A = R \quad (2)$$

Where R is an upper triangular matrix
and$H_j = M\{j\}$,$j = 1,2$, ... n are
orthogonal matrices such that

$$H_j H_j^T = I \ , j = 1\ ,2\ ,\ldots,n$$

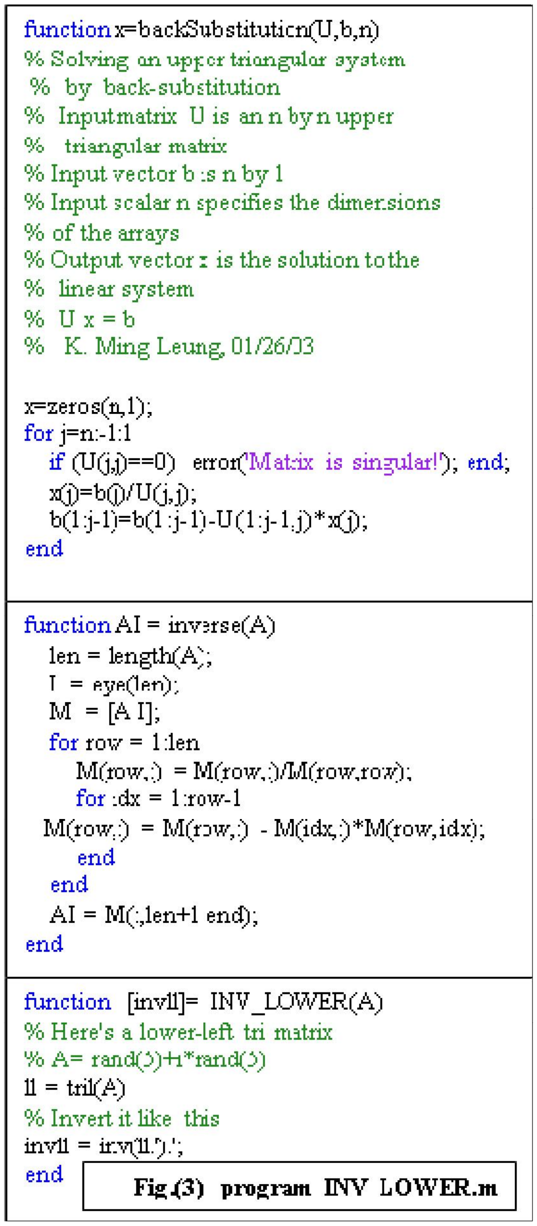
In the program Fig.(2)
$M\{j\}$,$j = 1,2$, ... $n$are cell arrays
because the algorithm used submatrix.As We are looking for large n , the algorithm uses a lot of matrixmultiplication of so many different sizes and even the last matrix H_n is nearly looks like an identity. Here, a question has been raised how these matrices be ordered to achieved less time complexity. In general this can be shown by what is known dynamic programming that willgiven in the next section D. Alternatively these cell matrices can be partitioned pair wisely taking care of identity matrices and rows of zeros as to be used in parallel techniques algorithm, especially for spare matrices.

2.4. dynamic programming

The idea emerged from the recursive divide and conquer algorithm called Stassen's algorithm of matrices for which the problem is divided into a smaller sub problems with operations carried on them. The technique is easily understood from a simple example .Consider the evaluations of the product of n matrices

$$M = M_1 * M_2 * M_3 \ldots\ldots\ldots * M_n \quad (3)$$

Where each matrix M_i with r_{i-1}rows and r_i
Columns . The order for which the matrices are multiplied together can be of a significant on the total number of operations required to evaluate M no matter which matrix algorithm is used . Now consider the multiplication



```
function x=backSubstitution(U,b,n)
% Solving an upper triangular system
 % by back-substitution
% Inputmatrix U is an n by n upper
%   triangular matrix
% Input vector b is n by 1
% Input scalar n specifies the dimensions
% of the arrays
% Output vector x is the solution to the
%  linear system
%  U x = b
%  K. Ming Leung, 01/26/03

x=zeros(n,1);
for j=n:-1:1
   if (U(j,j)==0)  error('Matrix is singular!'); end;
   x(j)=b(j)/U(j,j);
   b(1:j-1)=b(1:j-1)-U(1:j-1,j)*x(j);
end
```

```
function AI = inverse(A)
   len = length(A);
   I  = eye(len);
   M  = [A I];
   for row = 1:len
      M(row,:) = M(row,:)/M(row,row);
      for idx = 1:row-1
     M(row,:) = M(row,:) - M(idx,:)*M(row,idx);
      end
   end
   AI = M(:,len+1 end);
end
```

```
function  [invll]= INV_LOWER(A)
% Here's a lower-left tri matrix
% A= rand(5)+i*rand(5)
ll = tril(A)
% Invert it like  this
invll = inv(ll.').';
end
```

Fig.(3) program INV LOWER.m

where the dimensions of each M_i is shown in square brackets.
Evaluating M in the order$M = M_{1_{[10X20]}} * (M_{2_{[20x50]}}$
$* (M_{3_{[50x1]}} * M_{4_{[1x100]}}))$ (5)
requires 125000 operations in (5) while evaluating M in the order (4) requires 2200 operations. The algorithm is given below as Fig.(4)

dynamic algorithm:
dynamic programming algorithm for computing the minimum cost order of multiplying a string of n matrices

$$M = M_1 * M_2 * M_3 \ldots\ldots\ldots * M_n \quad (3)$$

Input :r_{0i} , r_1 ,r_2 , ... , r_n where r_{i-1} and r_i are the dimensions of the matrix M_i
Output : The minimum cost of multiplying $M_i's$, assuming pqr operations are required to multiply pxq matrix by qxr matrix
Method : it is shown in Fiq.(4) & Table 1

```
Begin
for i=1 until n do $m_{ii}$ = 0
    for l =1 until n-1 do
        for i=1 until n-l-- do  begin
            j=i+l
            $m_{ii} = \min_{i \le k < j} ( m_{i,k} + m_{k+1,j} + r_{i-1} * r_k * r_j$
        End
Write $m_{i,n}$
endendend
```

Fig.(4) Dynamic algorithm for ordering `matrix multiplication

Table 1: Cost of computing product $\boldsymbol{M = M_i * M_{i+1} * M_{i+2} \dots \dots \dots * M_j}$

m_{11}=0	m_{22}=0	m_{33}=0	m_{44}=0
m_{12}=10000	m_{23}=1000	m_{34}=5000	
m_{13}=1200	m_{24}=3000		
m_{14}=2200			

3. SPARE MATRICES AND OPERATIONS

There are several approaches for representation of matrices the easiest and not complicated are well known using two forms , one is known as rows representation for which the rows are contiguously laid which is called the Compressed Rows Representation (CRR) and the second is columns representation called Compressed Columns Representation (CCR).

3.1. Compressed Rows Representation (CRR)

Given the sparse matrix

$$A = \begin{array}{cccc} 2.0 & -1.0 & 0 & 0 \\ -1.0 & 2.0 & -1.0 & 0 \\ 0 & -1.0 & 2.0 & -1.0 \\ 0 & 0 & -1.0 & 2.0 \end{array}$$

Compressed Rows Representation (CRR) is given in Table 2

Table 2: Rows Representation (CRR)

pointers	n	col n	row n	Value n
	1	1	1	2.0
2	2	2	1	-1.0
	3	1	2	-1.0
	4	2	2	2.0
3	5	3	2	-1.0
	6	2	3	-1.0
	7	3	3	2.0
3	8	4	3	-1.0
	9	3	4	-1.0
2	10	4	4	2.0

sparse(A')

J I	
(1,1)	2
(2,1)	-1
(1,2)	-1
(2,2)	2
(3,2)	-1
(2,3)	-1
(3,3)	2
(4,3)	-1
(3,4)	-1
(4,4)	2

sparse(A)
answer as

(I ,J)	V
(1,1)	2
(2,1)	-1
(1,2)	-1
(2,2)	2
(3,2)	-1
(2,3)	-1
(3,3)	2
(4,3)	-1
(3,4)	-1
(4,4)	2

Fig.(5) sparse(A) &sparse(A')

Now if one apply the Matlabcommand **sparse** to the matrix A givenabove gets vector I , J and vector for the values V without the pointers given in Table 2((here the pointers shows the number of each elements in the row and can be useful for multiplying two vectors)) as in Fiq.(5).

Observe that the Matlab command **sparse(A)** represent matrices using CCR contiguously while sparse(A^T)using CRR contiguously whereA^T is the transpose of A. This means that when multiplying two Matrices A*B and to apply the sparse command one needs to find the spare(B^T) and algorithm to multiply two vectors. We will describe such an algorithm in section B .

3.2 Multiply two Sparse Vectors

There are different algorithms in the literature depending on the algorithm applied and the language such as NESL or C++ using parallel computing. Here ,is just a processing algorithm using the sparse command that is as in MATLAB ITS DESCRIPITION.

S = sparse(i,j,s,m,n,nzmax) uses vectors i, j, and s to generate an m-by-n sparse matrix such that S(i(k),j(k)) = s(k), with space allocated for nzmaxnonzeros. Vectors i, j, and s are all the same length. Any elements of s that are zero are ignored, along with the corresponding values of i and j. Any elements of s that have duplicate values of i and j are added together.

Algorithm to Multiply two spare matrices A and B ((B^T is in CRR))

The algorithm is in Fiq.(6) with following steps required :
Step1 : [m,n]=size(A); [Ia ,Ja,sa]=sparse(A)
Step2 : [n,p]=size(B); [Ib ,Jb,sb]=sparse(B')
Step3: write the function given in Fiq.(6) as it is STORESUM (C,q,r,col,sum)
Step4: delete **call** FAST –TRANSPOSE (B,B^T) line 4
Step5: write the algorithm as in Fiq.(6)

B. Operations on Spare Matrices

On matricesWe describe some operations

Reordering for Sparsely matrices

Reordering thecolumns of a matrix can often make its LU or QR factors sparser.Reordering the rows and columns can often make its Cholesky factors sparser. The simplest suchReordering thecolumns of a matrix can reordering is to sort the columns by nonzero count. This is sometimes a good reordering for matrices with very irregular structures, especially if there is great variation in the nonzero counts of rows or columns.
The function **p = colperm(S)** computes a permutation that orders the columns of a matrix by the number of nonzeros in each column from smallest to largest.

Reordering to Reduce Bandwidth

The reverse Cuthill-McKee ordering is intended to reduce the profile or bandwidth of the matrix. It is not guaranteed to find the smallest possible bandwidth, but it usually does. The function **symrcm(A)** actually operates on the nonzero structure of the symmetric matrix **A + A'**, but the result is also
useful for asymmetric matrices. This ordering is useful for matrices that come from one-dimensional problems or problems that are in some sense "long and thin."

Approximate Minimum Degree Ordering

The degree of a node in a graph is the number of connections to that node. This is the same as the number of off-diagonal nonzero elements in the corresponding row of the adjacency matrix. The approximate minimum degree algorithm function **symamd(A)** generates an ordering based on how these degrees are altered during Gaussian elimination or Cholesky factorization. It is a complicated and powerful algorithm that usually leads to sparser factors than most other orderings, including column count and reverse Cuthill-McKee. Because the keeping track of the degree of each node is very time-consuming, the approximate minimum degree algorithm uses an approximation to the degree, rather than the exact degree

Reordering and Factorization.

This example shows the effects of reordering and factorization on sparse matrices as in Fig.(7).

```
procedure MMULT (A, B, C)
// A is an m x n, B an n x p sparse matrix, C = A * B, an m x p matrix //
1   $(m,n,t_1) \leftarrow (A(0,1),A(0,2),A(0,3))$
2   if n = B(0,1)then $(p,t_2) \leftarrow (B(0,2),B(0,3))$
3           else[print ('incompatible matrices '):stop]
4   call FAST –TRANSPOSE (B,$B^T$)
5   $i \leftarrow q \leftarrow$ row_ begin $\leftarrow 1; r \leftarrow A(i,1)$;
6   $A(t_1+1,1) \leftarrow n+1; B^T(t_2+1,1) \leftarrow p+1$; sum$\leftarrow$ o; $B^T(t_2+1,2) \leftarrow 0$
7   while $i \le t_1$ do        // generate row r of C //
8       col $\leftarrow B^T(1,1)$; $j \leftarrow 1$
9       while $j \le t_2 + 1$ do       // multiply row r of A by column col of B//
10          case
11            $A(i,1) \ne r$:         // end of row r of A //
12                    call STORESUM (C,q,r,col,sum)
13                    $i \leftarrow$ row _begin
14                    while $B^T(j,1)$= col do      //go to next column of B //
15                    $j \leftarrow j+1$
16                    end
17                    col$\leftarrow B^T(j,1)$
18            : $B^T(j,1) \ne$ col:        //end Of column col of B//
19                    call STORESUM (C,q,r,col,sum)
20                    $i \leftarrow$ row _ begin ;col$\leftarrow B^T(j,1)$ //set to multiply row r with next column//
21            : $A(i,2) \le B^T(j,2)$ :       //advance to next term in row r//
22                      $i \leftarrow i+1$
23            : $A(i,2)=B^T(j,2)$ :            //add to sum//
24                    sum$\leftarrow$ sum $+A(j,3)*B^T(j,3)$
25              $i \leftarrow i+1; j \leftarrow j+1$
26              :else           //advance to next term in column col//
27                      $j \leftarrow j+1$
28          end

Procedure STORESUM (C,q,r,col,sum)
//if sum is nonzero then along with its row and column position ,it is stored into q-th entry of the matrix//
  if sum $\ne 0$ then [ (C(0,1),C(0,2),C(0,3)) $\leftarrow$(row,col,sum) ; $q \leftarrow q+1$ ; sum$\leftarrow 0$ ]
 end STORESUM
```

Fig.(6) Algorithm to Multiply two spare matrices A and B ((B^T is in CRR))

If you obtain a good column permutation p that reduces fill-in, perhaps from **symrcm** or **colamd**, then computing **lu(S(:,p))** takes less time and storage than computing lu(S).The number of **nonzero**s in that matrix is a measure of the time and storage required to solve linear systems involving B. Here are the nonzero counts for the three permutations being considered.

- lu(B) (Original): 1022
- lu(B(r,r)) (Reverse Cuthill-McKee): 968
- lu(B(m,m)) (Approximate minimum degree): 636

```
function symrcmspy()
B = bucky+4*speye(60);
r = symrcm(B);
p = symamd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R,4), title('B(r,r)')
subplot(2,2,2), spy(S,4), title('B(s,s)')
subplot(2,2,3), spy(chol(R),4), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S),4), title('chol(B(s,s))')
end
&&&&&&&&&&&&&&&&&&&&&&&&&&&
function symrcmspy1()

figure
B = bucky;
B = B - 3*speye(size(B));
r = symrcm(B);
m = symamd(B);
subplot(1,3,1)
spy(lu(B))
title('Original')

subplot(1,3,2)
spy(lu(B(r,r)))
title('Reverse Cuthill-McKee')

subplot(1,3,3)
spy(lu(B(m,m)))
title('Approx Min Degree')
```

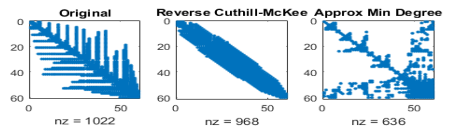


Fig.(7) shows the nonzeros counts as a measure for matrix B=bucky and the operation symrcm and symamd. Figure from file symrcmspy1.m shows the scattering on the diagonal or fill-in .

- In fact there are many direct factorizations Matlab function for spare such as **qr , lu and ilu** and for iterative techniques as Conjugate gradient squared ((**cgs**)) and Preconditioned conjugate gradient((**pcg**)).

4. VEIWS ON PARALLEL COMPUTING AND MATRIX MULTIPLICATION

Parallel Computing is a subject used to compute intensive data with the goal to economize both the time and the storage that usually require SUPPER or PARALLEL machine computing ((i.e. MATLAB Parallel Computing Toolbox installed on your desktop computer and have configured it for use at OHIO Supercomputer Center OSC)).Instead , computers with four cores or more can be connected in a network topology **ring** ,**tours** ,**mesh** or **hypercube** to work in parallel. These cores are called workers((processors)) each have an index number ((**labindx**)) and the total number of processors called ((**numlabs**)) , Alternatively , these processers can be connected as shared memory system as in Fig.(8).

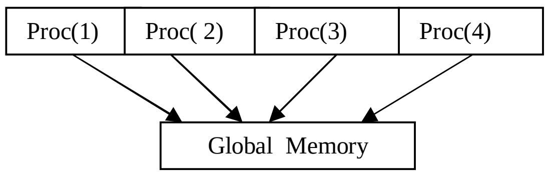


Fig 8: 4-Processors shared memory system

Option 1: MPI parallelism

```
pmode open
% start parallel mode with 2  lab workers
D = distributed.rand(3); % distributed across
        workers of matlabpool
R = D * D % mtimes overloaded to compute
                in parallel
R =  0.3165    0.9958    0.7154
       0.0097    0.4595    0.3512
       0.0149    1.2476    0.9831

R =   0.7074    0.3176    0.3263
       0.3282    0.6694    0.2251
       0.5815    0.2582    0.3707
```

Option 2: GPU parallelism

```
G = gpuArray(rand(2000)); %   place data
                        on the GPU
     G2 = G * G; % operate on it in  parallel
```

Fig.(9) explicit options on parallelism

However , there are two explicit approaches clarify the meaning of parallel computing , **Option 1: MPI parallelism** and **Option 2: GPU parallelism** that presented in Fig.(9). In option two , We find an ERORR indicating the message" The CUDA driver could not be loaded. The library name used was 'nvcuda.dll'. The error was: The specified module could not be found " . In option 1 ,M.P.I. Message Passing Interface , the command distributed.rand (3) distributes a generated matrix random number of dimension

3 ,each is different across **two** labs worker.The command distributed and codistributed are important for parallel matrix multiplication as We can see in Fig.(9).

4.1 Fox's Algorithm for Matrix Multiplication

.The idea of Fox's algorithm is to multiply in

Parallel two matrices $C_{mXn} = A_{mXp}B_{pXn}$ (6)

this is often done by dividing a matrix into blocks (often called tiles these days).The obvious plan of attack here is to break the matrices into blocks, and then assign different blocks to different MPI nodes. Assume that $\sqrt{p}$ evenly divides n, and partition each matrix intosubmatrices of size $\frac{n}{\sqrt{p}}$ x $\frac{n}{\sqrt{p}}$. In other words, each matrix will be divided into m rows and m columns of blocks, where m = $\frac{n}{\sqrt{p}}$

One of the conditions assumed here is that the matrices A and B are stored in a distributed manner across the nodes. This situation could arise for several reasons:

* The application is such that it is natural for each node to possess only part of A and B.
* One node, say node 0, originally contains all of A and B, but in order to conserve communication time, it sends each node only parts of those matrices.
* The entire matrix would not fit in the available memory at the individual nodes

Consider the node that has the responsibility of calculating block (i,j) of the product C, which it cab shown to be calculated as

$$\sum_{k=0}^{m-1} A_{i;(i+k)mod\ m}\, B_{(i+k)mod\ m\ ;j} \qquad (7)$$

Eqn.(7) , starting from A_{ii} , then go across row i of A , wrapping back up to the left end when you reach the right end. The order of summation in this rearrangement will be the actual order of computation. The algorithm is in Fig.(10) ,then as follows. The node which is handling the computation of Cij does this (in parallel with the other nodes which are working with their own values of i and j):

```
1  iup = i+1 mod m;
2  idown = i-1 mod m;
3  for ( k = 0 ; k < m; k++) {
4    km = ( i+k ) mod m;
5    broadcast (A[ i ,km] ) to all nodes
        handling row i o f C;
6    C[ i , j ] = C[ i , j ] + A[ i ,km] *B[km, j ]
7    send B[km, j ] to the node
        handling C[ idown , j ]
8    receive  new B[km+1 mod m, j ]
     from the node handling C[ iup , j ]
9        }
```

Fig.(10) Fox's Algorithm to Multiply 2 Matrices

4.2 Matrix Multiply in CUDA

This is not a public program and must be invoke by Kernel Invocation (Host-side Code) . The program in Fig.(11**)**

```
// Setup the execution configuration
// TILE_WIDTH is a #define constant
dim3 dimGrid(Width/TILE_WIDTH,
                    Width/TILE_WIDTH, 1);
dim3 dimBlock(TILE_WIDTH,
                              TILE_WIDTH, 1);
// Launch the device computation threads!
    MatrixMulKernel<<<dimGrid,
          dimBlock>>>(Md, Nd, Pd, Width);
```

```
// Matrix multiplication kernel –per thread code
__global__ void MatrixMulKernel(float* d_M,
                float* d_N, float* d_P, int Width)
{
// Pvalue is used to store the element of the matrix
// that is   computed by the thread
float Pvalue = 0;
__global__ void MatrixMulKernel(float* d_M,
                 float* d_N, float* d_P, intWidth)
{
// Calculate the row index of the d_P element and d_M
intRow = blockIdx.y*blockDim.y+threadIdx.y;
// Calculate the column idenxof d_Pand d_N
intCol = blockIdx.x*blockDim.x+threadIdx.x;
if ((Row < Width) && (Col < Width)) {
float Pvalue= 0;
// each thread computes one element of the block
sub-matrix
for (intk = 0; k < Width; ++k)
Pvalue+= d_M[Row*Width+k] *d_N[k*Width+Col];
d_P[Row*Width+Col] = Pvalue;
}
}
```

Fig.(11) Matrix Multiplication using CUDA

```
function  redistributed()
A =reshape(1:16,4,4)
DIST=codistributor2dbc([2,1],2);
D=codistributed(A,DIST);
L=getLocalPart(D)
codist = codistributor1d(1,[2 2] ,[4 4]);
X = codistributed.build(L,codist)
DD=getLocalPart(X)
end
A = 1  5  9   13          A =  1  5  9   13
    2  6  10  14                2  6  10  14
    3  7  11  15                3  7  11  15
    4  8  12  16                4  8  12  16


L = 1  5  9  13         L = 3  7  11  15
    2  6  10  14            4  8  12  16


 This worker stores X(1:2,:).  This worker stores X(3:4,:).
 LocalPart: [2x4 double]         LocalPart: [2x4 double]
              Codistributor: [1x1 codistributor1d]


 DD = 1  5  9  13           DD = 3  7  11  15
        2  6  10  14                4  8  12  16
```

Fig.(12) The file redistributed.m

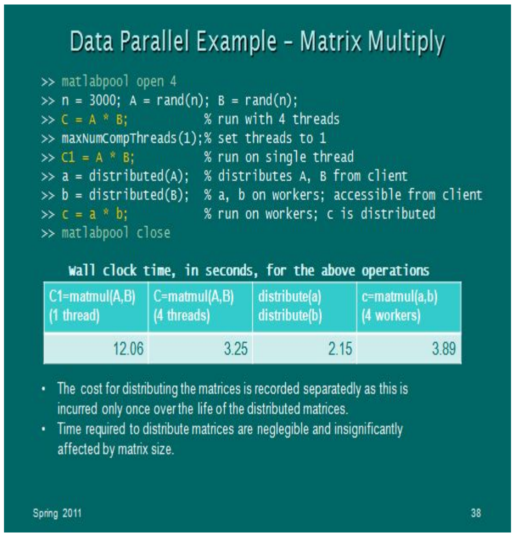
4.3. Multiple Approaches to Distribute arrays

Distributing an array means to subdivided into submatrices so that one can compute with blocks and multiple threading .Multiple threading means an implicit parallel computing

In Matlab programming arrays are distributed across the workers by applying the commands codistributor2dbc , codistributor1d ,etc ,.., and codistributed.build. We demonstrate this in file **redistributed.m** Fig.(12) on two labs workersand gathering these segements by the command codistributed.build.

It is well known , a parallel operation is worthwhile only if the array size is sufficiently large to gain significant parallel speedup to offset the overhead cost. Note that the overhead cost is incurred only once while the distributed arrays may be used many times. Note also that there is always a communication cost whenever the distributed array need to be transferred among the workers or with the MATLAB workspace .

There are multiple ways to distribute arrays. You can distribute an array among the workers directly from the MATLAB workspace



Data Parallel Example – Matrix Multiply

```
>> matlabpool open 4
>> n = 3000; A = rand(n); B = rand(n);
>> C = A * B;             % run with 4 threads
>> maxNumCompThreads(1);% set threads to 1
>> C1 = A * B;            % run on single thread
>> a = distributed(A);  % distributes A, B from client
>> b = distributed(B);  % a, b on workers; accessible from client
>> c = a * b;             % run on workers; c is distributed
>> matlabpool close
```

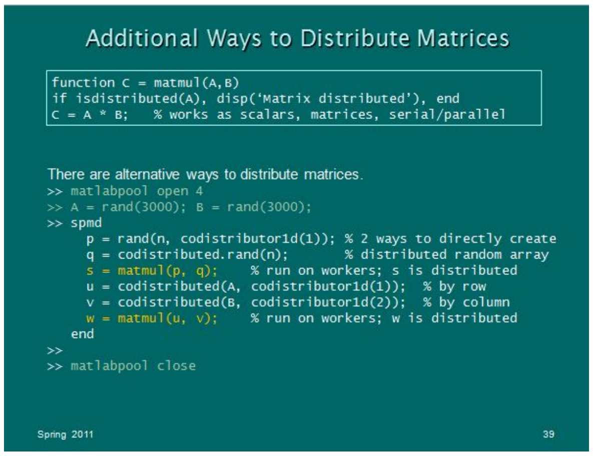
Wall clock time, in seconds, for the above operations

C1=matmul(A,B) (1 thread)	C=matmul(A,B) (4 threads)	distribute(a) distribute(b)	c=matmul(a,b) (4 workers)
12.06	3.25	2.15	3.89

- The cost for distributing the matrices is recorded separatedly as this is incurred only once over the life of the distributed matrices.
- Time required to distribute matrices are neglegible and insignificantly affected by matrix size.

Spring 2011 38

with ***distributed*** (see above slide) or from the ***spmd*** environment with ***codistributed*** (see slide below.


Additional Ways to Distribute Matrices

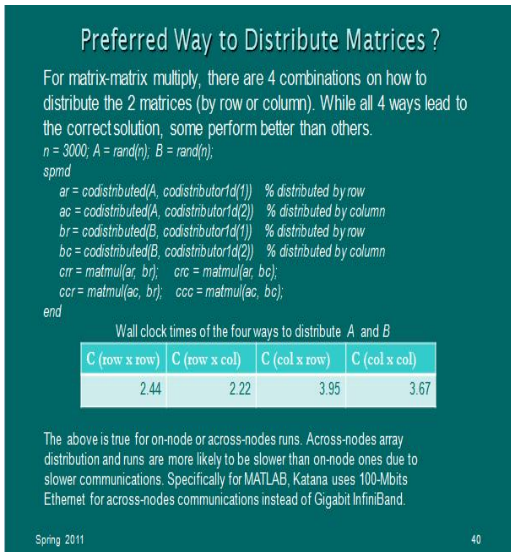
```
function C = matmul(A,B)
if isdistributed(A), disp('Matrix distributed'), end
C = A * B;   % works as scalars, matrices, serial/parallel
```

There are alternative ways to distribute matrices.

```
>> matlabpool open 4
>> A = rand(3000); B = rand(3000);
>> spmd
      p = rand(n, codistributor1d(1)); % 2 ways to directly create
      q = codistributed.rand(n);        % distributed random array
      s = matmul(p, q);    % run on workers; s is distributed
      u = codistributed(A, codistributor1d(1));  % by row
      v = codistributed(B, codistributor1d(2));  % by column
      w = matmul(u, v);    % run on workers; w is distributed
   end
>>
>> matlabpool close
```

Spring 2011 39

MATLAB constructs *zeros*& *rand* have been over loaded to handle parallel distribution. Decomposing an array directly with constructor function not only saves memory, it may be more efficient well.


Preferred Way to Distribute Matrices ?

For matrix-matrix multiply, there are 4 combinations on how to distribute the 2 matrices (by row or column). While all 4 ways lead to the correct solution, some perform better than others.

```
n = 3000; A = rand(n); B = rand(n);
spmd
   ar = codistributed(A, codistributor1d(1))   % distributed by row
   ac = codistributed(A, codistributor1d(2))   % distributed by column
   br = codistributed(B, codistributor1d(1))   % distributed by row
   bc = codistributed(B, codistributor1d(2))   % distributed by column
   crr = matmul(ar, br);    crc = matmul(ar, bc);
   ccr = matmul(ac, br);    ccc = matmul(ac, bc);
end
```

Wall clock times of the four ways to distribute *A* and *B*

C (row x row)	C (row x col)	C (col x row)	C (col x col)
2.44	2.22	3.95	3.67

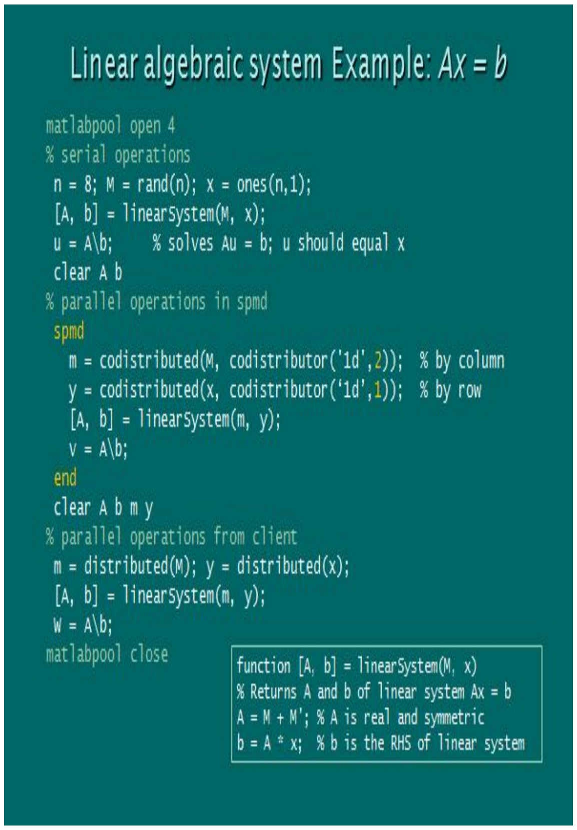
The above is true for on-node or across-nodes runs. Across-nodes array distribution and runs are more likely to be slower than on-node ones due to slower communications. Specifically for MATLAB, Katana uses 100-Mbits Ethernet for across-nodes communications instead of Gigabit InfiniBand.

Spring 2011 40

According to the above timing data, distributing *a* by row and *b* by column yield the most efficient matrix multiply operation. This is a direct consequence of matrix multiply rules. For other applications, the optimal choice of matrix distribution may be less obvious or straightforward unless you are familiar with the underlining algorithm used. The linear algebraic system solver, shown next, is just such an example.

- Linear algebraic system example ($Ax = b$)

Similar to matrix multiply, linear solver ($Ax=b$) operations are performed in parallel since the matrices involved are distributed


Linear algebraic system Example: $Ax = b$

```
matlabpool open 4
% serial operations
 n = 8; M = rand(n); x = ones(n,1);
 [A, b] = linearSystem(M, x);
 u = A\b;     % solves Au = b; u should equal x
 clear A b
% parallel operations in spmd
 spmd
   m = codistributed(M, codistributor('1d',2));  % by column
   y = codistributed(x, codistributor('1d',1));  % by row
   [A, b] = linearSystem(m, y);
   v = A\b;
 end
 clear A b m y
% parallel operations from client
 m = distributed(M); y = distributed(x);
 [A, b] = linearSystem(m, y);
 W = A\b;
matlabpool close
```

```
function [A, b] = linearSystem(M, x)
% Returns A and b of linear system Ax = b
A = M + M'; % A is real and symmetric
b = A * x;  % b is the RHS of linear system
```

5. COMPUTATIONAL REMARK

Parallel computing is not an easy subject which is differ very much from serial computing for which there are many reasons and factor need to be analysis and understood first , especially if one has not got a super computer. Each instruction such as SPMD , PARFOR or others need to be handled carefully ,e,g. getlocalpart will give an error , Matlab dose not recognize it.

IV. ACKNOWLEDGEMENT

I would to thank u.k., Wales University during 1979-1984 to be awarded my M.SC.and Ph.D.This due I have got the references [1]&[2]

REFERENCES

[1] Aho|Hopcroft|Ullman . The Design and Analysis of Computer Algorithms.Addison-Wesley Publishing Company,1974

[2] E.Horowitz &S. Sahni, Fundamental of Data Structures. PITMANN PUBLISHING Limited ,1977
[3] D.A.Gismalla , Matlab Software for Iterative Methods and Algorithms to Solve a Linear System , International Journal of Engineering and Technical Research (IJETR) , ISSN :2321-0869,Volume-2 Issue-2, Februray 2014 http://www.erpublication.org/IJETR/vol_issue.php?abc1=20
[4] Akshay Panajwar,Prof.M.A.Shah Efficient Solver for Linear Algebraic Equations on Parallel Architecture using MPI . 6th International Conference on Computer Science and Information Technology (ICCIT 2015) ISBN: 978-93- 85225-32-1, 31st May 2015, Pune
[5] Working with Distributed Arrays . Boston University Information Sciences &Technology
http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/distributed-array-examples
[6] Implicit Parallelism (Multithreading).Boston University Information Sciences & Technology
http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/implicit-parallelism
[7] Parallel and GPU Computing Tutorials
https://www.mathworks.com/videos/parallel-computing-tutorial-deeper-insights-into-using-parfor-4-of-9-91566.html
[8] Working with Codistributed Arrays
https://www.mathworks.com/help/distcomp/working-with-codistributed-arrays.html#bqjuvpl
[9] SuiteSparse is a Suite for Sparse Matrices Algogithm
http://www.cise.ufl.edu/research/sparse
[9] Configuring the MATLAB Parallel Computing Toolbox at OSC | Ohio
https://www.osc.edu/research/hll/matlab
[10] [Parallel MATLAB: Single Program Multiple Data - Interdisciplinary
www.icam.vt.edu/Computing/fdi_2012_spmd.pdf
[11] Mondriaan for sparse matrix partitioning
http://www.staff.science.uu.nl/~bisse101/Mondriaan/