8-2013

Distributed Sparse Matrix Solver and Pivoting Algorithms for Large Linear Equations

Esteban Torres
University of Texas-Pan American

DISTRIBUTED SPARSE MATRIX SOLVER AND PIVOTING ALGORITHMS FOR LARGE LINEAR EQUATIONS

A Thesis

by

ESTEBAN TORRES

Submitted to the Graduate School of the
University of Texas-Pan American
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2013

Major Subject: Electrical Engineering

DISTRIBUTED SPARSE MATRIX SOLVER AND PIVOTING ALGORITHMS

FOR LARGE LINEAR EQUATIONS

A Thesis
by
ESTEBAN TORRES

COMMITTEE MEMBERS

Dr. Yul Chu
Chair of Committee

Dr. Jae Son
Committee Member

Dr. Andras Balogh
Committee Member

Dr. Maria C. Villalobos
Committee Member

August 2013

ABSTRACT

Torres, Esteban, Distributed Sparse Matrix Solver and Pivoting Algorithms for Large Linear Equations. Master of Science (MS), August, 2013, 160 pp., 4 tables, 30 illustrations, references, 28 titles.

This thesis presents a distributed sparse direct solver and pivoting strategies for distributed sparse LU factorization. In Chapter I, we introduce some background concepts in linear algebra. In Chapter II, we discuss parallel hardware architectures and introduce our problem for discussion. In Chapter III, we describe the implementation of our sparse direct solver and our pivoting algorithms. Next, we present our simulation methodology and results in Chapter IV and we show that our pivoting algorithm yields up to 50% more accurate results than a current state-of-the-art solver, SuperLU_DIST. Finally, in Chapter V, we present some ideas to further improve the accuracy and speed of our distributed sparse matrix solver.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

Solving linear equations is a routine task in most engineering disciplines. Circuit analysis, image processing, and computational fluid dynamics, for example, rely heavily on solving linear systems of equations [1]. In many cases, these equations take the form $Ax=b$, where A is a matrix, b is a vector, and x is the vector of unknowns [2]. Several factors can influence the computation time for linear equations. When A is a dense matrix, the computation time mainly depends on the size of the matrix. If A is sparse, the computation time can be influenced by the size of the matrix, the number of non-zeros, or sparsity, and the numerical structure of the matrix [2]. As modern engineering systems grow more and more complex, there is a need to solve bigger and more complex linear equations. Unfortunately, the speed of modern computer systems has reached a limit due to constraints in material properties [3]. In order to continue improving the performance, there has been a shift to multi-core and distributed systems [3][4]. Exploiting this new hardware, however, also requires the development of new algorithms so that improvements in performance do not sacrifice numerical stability.

1.1 Solving Linear Equations

1.1.1 Circuit Simulations

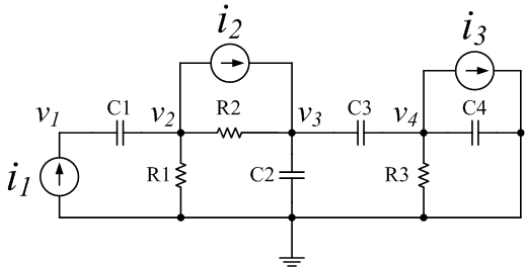
In the field of electrical engineering, circuit analysis involves formulating and solving sets of linear equations from circuit schematics [5]. When solving these sets of equations by

hand, the most common method used is Gaussian elimination where equations are manipulated using row operations until the final set of equations is in row echelon form, as shown in Figure 1. The set of equations can then be solved by simple back substitution.

$$
\begin{aligned}
8\,x_1 + 3\,x_2 + 7\,x_3 + 7\,x_4 + 9\,x_5 &= 187 \\
1\,x_2 + 3\,x_3 + 7\,x_4 + 2\,x_5 &= 59 \\
5\,x_3 + 4\,x_4 + 6\,x_5 &= 69 \\
3\,x_4 + 5\,x_5 &= 19 \\
9\,x_5 &= 18
\end{aligned}
$$

Figure 1 – *Row echelon form*

Rather than solving equations by hand, circuit designers frequently use circuit simulation software, such as PSPICE [6], to perform circuit analysis. The software formulates equations from a given circuit, as shown in Figure 2, and solves the system automatically. In many instances, it is necessary to solve the system in rapid succession to analyze the circuit's transient response or frequency response. Therefore, it's important that the software be able to solve the system as quickly as possible.



$$
\begin{bmatrix}
C1\frac{dv}{dt} & -C1\frac{dv}{dt} & 0 & 0 \\
-C1\frac{dv}{dt} & \frac{1}{R1}+\frac{1}{R2}+C1\frac{dv}{dt} & -\frac{1}{R2} & 0 \\
0 & -\frac{1}{R2} & \frac{1}{R2}+C2\frac{dv}{dt}+C3\frac{dv}{dt} & -C3\frac{dv}{dt} \\
0 & 0 & -C3\frac{dv}{dt} & \frac{1}{R3}+C3\frac{dv}{dt}+C4\frac{dv}{dt}
\end{bmatrix}
\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}
=
\begin{bmatrix} i_1 \\ -i_2 \\ i_2 \\ -i_3 \end{bmatrix}
$$

Figure 2 – *Sample circuit and its corresponding matrix equation*

1.1.2 Iterative and Direct Solving Methods

There are many methods that can be used to solve linear equations. Direct methods, such as LU decomposition and Cholesky decomposition, arrive at a solution using a finite and predictable number of operations. In contrast, iterative methods make an approximation of the solution and repeatedly refine the approximation until an acceptable solution is reached [2]. Iterative methods are specially designed to exploit the mathematical properties of certain problems such as “finite difference or finite element models of physical systems” [2].For those problems, iterative methods can be much faster than direct methods [7]. However, a correct solution cannot be guaranteed for all matrices [8]. Therefore, the main weakness of iterative methods is that they are not as robust as direct methods [8].

One of the most widely used direct methods is LU decomposition, which is based on Gaussian elimination [2][7]. LU decomposition is frequently used by mathematical software, such as SuperLU_DIST [9], PARDISO [10], and WSMP [11], to solve systems from many different disciplines. Researchers are actively looking for ways to improve the performance and numerical stability of LU decomposition for solving large equations on multi-core and distributed systems.

1.2 Dense LU Decomposition

Solving the system $Ax=b$using LU decomposition involves several steps. First, the coefficient matrix is factored so that $A=LU$ [1] as shown in Figure 3. The pseudo-code, shown in Figure 4, illustrates how the factors, L and U, are computed. The diagonal elements are factored first, followed by the elements below and to the right of the diagonal.

The solve phase involves two triangular equations. First, the lower triangular equation $Ly=b$ is solved to get y. Finally, the upper triangular equation $Ux=y$ is solved to get x. The

forward and back substitution phases are illustrated in Figure 5 and a sample factorization, *A=LU*, is shown in Figure 6.

$$\underset{A}{\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}} = \underset{L}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix}} \underset{U}{\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}}$$

Figure 3 – *Numerical structure of LU decomposition*

Let *A* be a matrix of size m-by-n

```
for i=1 to m
     Lii = 1
     Uii = A_ii − Σ_{k=i−1}^{k=1} (L_{i,k} · U_{k,i})
     for j=i+1 to m
          Lji = (A_ji − Σ_{k=i−1}^{k=1} (L_{j,k} · U_{k,i})) / U_ii
     endfor
     for j=i+1 to n
          Uij = A_ij − Σ_{k=i−1}^{k=1} (L_{i,k} · U_{k,j})
     endfor
endfor
```

Figure 4 – *Pseudo-code for dense LU decomposition*

Forward Substitution

$$\underset{L}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix}} \underset{y}{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}} = \underset{b}{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}}$$

Back Substitution

$$\underset{U}{\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}} \underset{x}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}} = \underset{y}{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}}$$

Figure 5 – *Dense matrix forward and back substitution*

```
A
  8.0000    3.0000    2.0000    4.0000    3.0000
  9.0000    3.0000    6.0000    1.0000         0
  1.0000    1.0000    7.0000    1.0000    9.0000
  5.0000    6.0000    6.0000    7.0000    4.0000
  2.0000    5.0000    4.0000    2.0000    9.0000

L
  1.0000         0         0         0         0
  1.1250    1.0000         0         0         0
  0.1250   -1.6667    1.0000         0         0
  0.6250  -11.0000    3.5385    1.0000         0
  0.2500  -11.3333    3.5385    1.3085    1.0000

U
  8.0000    3.0000    2.0000    4.0000    3.0000
       0   -0.3750    3.7500   -3.5000   -3.3750
       0         0   13.0000   -5.3333    3.0000
       0         0         0  -15.1282  -45.6154
       0         0         0         0   19.0712
```

Figure 6 – *Sample LU factorization*

1.3 Partitioned Dense LU Decomposition

In order to solve large matrix equations efficiently, it is important to develop algorithms that break the problem into smaller sections. Algorithms such as [12] work by partitioning matrices into sub-matrices and decomposing each sub-matrix individually. The advantage of these algorithms is that they help distribute computational load and make efficient use of memory. The pseudo-code in Figure 7shows how a dense matrix would be decomposed using partitioned dense LU decomposition. The algorithm works by factorizing the diagonal sub-matrix first, followed by the sub-matrices below and to the right of the diagonal sub-matrix.

Let *A* be a block matrix of size M-blocks by N-blocks.

```
for i=1 to M
      Factorize block A_ii into L_ii and U_ii
      for j=i+1 to M
         Factorize block L_ji
      endfor
      for j=i+1 to N
         Factorize block U_ij
      endfor
endfor
```

Figure 7 – *Pseudo-code for partitioned dense LU decomposition*

1.4 Numerical Stability and Pivoting Strategies

An algorithm that can generate accurate results 100% of the time is said to have high numerical stability [7]. Unfortunately, the numerical structure of a matrix can have a negative effect on the speed and accuracy that can be achieved with LU decomposition.

One of the problems that occur with LU decomposition is the notion of fill-in [2][7]. Depending on the structure of the matrix, elements that were zero in *A* may become non-zero in *L* and *U*. This can reduce the speed of LU decomposition since the extra non-zeros require extra floating point operations. Figure 8 shows the worst case and best case scenarios for fill-in during LU decomposition. This problem can be solved in general by pre-conditioning the matrix using graph-based algorithms and minimum degree algorithms as described in [13].

A=

```
  4   7   7   4   1
  7   8   0   0   0
  9   0   1   0   0
  9   0   0   7   0
  5   0   0   0   5
```

L=

```
1.0000       0       0       0       0
1.7500  1.0000       0       0       0
2.2500  3.7059  1.0000       0       0
2.2500  3.7059  0.9674  1.0000       0
1.2500  2.0588  0.5374  0.0407  1.0000
```

U=

```
4.0000   7.0000   7.0000   4.0000   1.0000
     0  -4.2500 -12.2500  -7.0000  -1.7500
     0        0  30.6471  16.9412   4.2353
     0        0        0   7.5528   0.1382
     0        0        0        0   5.0712
```

Worst Case

A=

```
  5   0   0   0   5
  0   8   0   0   7
  0   0   1   0   9
  0   0   0   7   9
  1   7   7   4   4
```

L=

```
1.0000       0       0       0       0
     0  1.0000       0       0       0
     0       0  1.0000       0       0
     0       0       0  1.0000       0
0.2000  0.8750  7.0000  0.5714  1.0000
```

U=

```
5.0000       0       0       0   5.0000
     0  8.0000       0       0   7.0000
     0       0  1.0000       0   9.0000
     0       0       0  7.0000   9.0000
     0       0       0       0 -71.2679
```

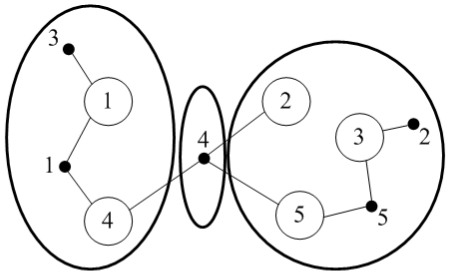
Best Case

Figure 8 – *Worst and best case for fill-in during LU decomposition*

Nested dissection is a commonly used graph-based algorithm that reduces fill-in during LU factorization [14]. This method works on a graph that represents the correlation between the rows and columns of the matrix. By cutting the graph, the matrix can be re-ordered so that fill-in is reduced and elements are grouped closer together. An example of nested dissection is shown

in Figure 9.The figure shows a column-net hypergraph [14]. The small dots are called hyperedges and represent columns in the matrix. The large numbered circles are called vertices and represent rows in the matrix.

Minimum degree algorithms, such as Markowitz [15] or approximate minimum degree [16], can also be used in addition to nested-dissection to further reduce fill-in. Other more advanced algorithms such as constrained column approximate minimum degree [17] can even be applied to sections of a matrix in order to optimize each section separately.



Original Matrix

2	0	7	0	0
0	0	0	6	0
0	5	0	0	1
4	0	0	7	0
0	0	0	6	2

Reordered Matrix

2	7	0	0	0
4	0	0	0	7
0	0	0	0	6
0	0	5	1	0
0	0	0	2	6

Figure 9 – *Example of nested dissection*

Another problem that occurs with LU decomposition is when zeros appear in the diagonal of *A* or *U*. This problem would cause LU decomposition to fail due to division by zero [2]. One way to prevent this is to pre-condition the matrix as described in [13]. However, matrix pre-conditioning is not 100% effective. Another method of solving this problem is to change the zero in the diagonal to a small number. This method, called perturbation [2][13], can help solve some systems but it can also lead to errors in the final solution. A third way to solve this problem

is to reorder the matrix as it is being factored. This method is called pivoting and there are several variations such as partial pivoting, threshold pivoting, and full pivoting [2][7].

Figure 10 shows an example of LU decomposition with partial pivoting. In the example, the last two rows of matrix *A* have been swapped in order to prevent a zero in the third diagonal element of *U*.

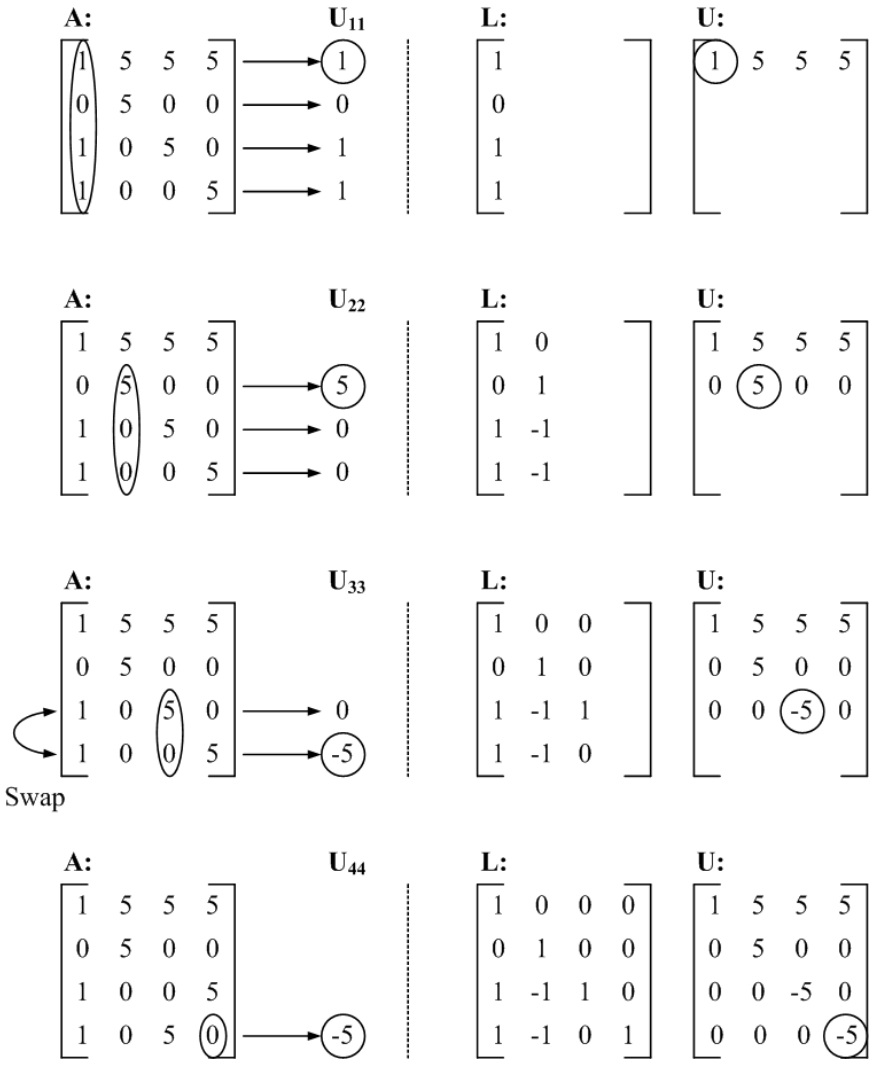


Figure 10 – *Example of LU decomposition with partial pivoting*

A third problem that occurs in LU decomposition is rounding errors [2][7]. This problem occurs when the diagonal element much smaller than the adjacent elements in the matrix, as

shown in Figure 11. Matrix pre-conditioning can help prevent this problem but a more robust method is to use pivoting during factorization. The main problem with pivoting is that in distributed memory systems, obtaining the best pivot element may require swapping rows or columns between two machines across the network. The amount of extra communication between nodes would result in slow system performance. In shared memory systems, pivoting may involve matrix elements that are not in system memory and must be fetched from the disk. This would lead to many disk operations and would also result in slow system performance. Instead of traditional pivoting algorithms, distributed and shared memory solvers must use different techniques as described in Chapter II.

```
A=
 1.32e-07  7.04e+04  3.81e+06  4.31e+05  8.34e+03
 6.44e+08  7.40e+00  5.61e+00  2.64e+00  1.07e+00
 4.54e+06  7.76e+00  2.10e+00  8.90e+00  2.21e-01
 3.88e+04  8.55e+00  2.60e+00  1.06e+00  5.05e+00
 6.18e+01  5.25e+00  4.82e+00  3.11e+00  2.69e+00

b=
 4.3197400000001322e+06
 6.4400001672000003e+08
 4.5400189809999997e+06
 3.8817260000000002e+04
 7.7670000000000002e+01

L=
 1.00e+00         0         0         0         0
 4.88e+15  1.00e+00         0         0         0
 3.44e+13  7.05e-03  1.00e+00         0         0
 2.94e+11  6.02e-05 -2.43e+02  1.00e+00         0
 4.68e+08  9.60e-08 -1.33e+02  5.51e-01  1.00e+00

U=
 1.32e-07  7.04e+04  3.81e+06  4.31e+05  8.34e+03
        0 -3.43e+20 -1.86e+22 -2.10e+21 -4.07e+19
        0         0  2.10e+00  8.90e+00  3.22e+01
        0         0         0  2.10e+03  7.82e+03
        0         0         0         0 -2.06e+01
Theoretical          Experimental
x=                     x=
   1.00                    1.00
   1.00                  242.05
   1.00                   -1.44
   1.00                   -7.97
   1.00                 -454.95
```

Figure 11 – *Rounding errors in LU decomposition*

CHAPTER II

DISTRIBUTED SYSTEMS AND PARALLEL SPARSE SOLVERS

Developing accurate and efficient algorithms requires in-depth knowledge of the underlying hardware and its limitations.

2.1 Parallel Hardware Architectures

Parallel systems can be classified into two categories: shared memory systems and distributed memory systems. Today, the majority of computers sold are multi-core systems which are classified as shared memory systems [3][4]. The memory hierarchy of shared memory systems consists of the L1 caches, L2 cache, system memory, and disk, as shown in Figure 12. The L1 cache is the fastest memory component but it is also the smallest and most expensive. The disk is the slowest but it is the least expensive and holds the most capacity.

Some shared memory solvers, such as [18], aim at solving large equations using machines with limited memory by efficiently moving matrix elements to and from the disk as needed. These solvers require special pivoting strategies to achieve good numerical stability while avoiding many read and write operations to disk.

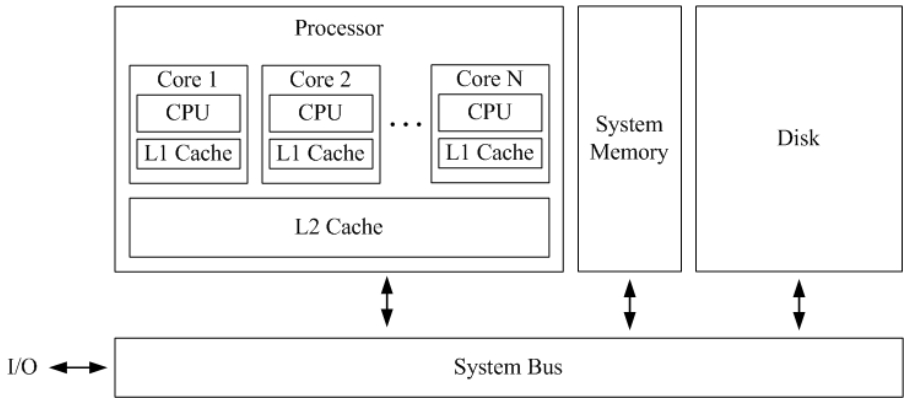


Figure 12 – *Shared Memory (multi-core) architecture [19]*

Programming in a shared memory system can be done with the OpenMP API [20]. The API allows any program to be converted into a parallel program by adding directives that specify which parts of the program can be executed in parallel. As the program is executed, the operating system creates a thread for each of the parallel sections. Each thread then updates the appropriate variables. Figure 13shows what a sample OpenMP program would look like using pseudo-code. In the figure, one parallel section computes the lower matrix *L* and another parallel section computes the upper matrix *U*. Since *j* is not declared as a shared variable, the value of *j* may be different in each of the parallel sections. Based on the pseudo-code in Figure 13, Figure 14 shows the matrix elements that would be required to be in memory in order to factorize the third row and column of the matrix in parallel.

Let A be a matrix of size m-by-n

```
for i=1 to m
   Lii = 1
   Uii = Aii − ∑_{k=i−1}^{k=1} (Li,k · Uk,i)
   #pragma omp parallel sections shared(A,L,U,i)
   {
      #pragma omp section
      {
         for j=i+1 to m
               Lji = (Aji − ∑_{k=i−1}^{k=1} (Lj,k · Uk,i)) / Uii
```

```
         endfor
      }
      #pragma omp section
      {
         for j=i+1 to n
            U_ij = A_ij − ∑_{k=i−1}^{k=1} (L_{i,k} · U_{k,j})
         endfor
      }
   }
endfor
```

Figure 13 – *Pseudo-code for LU decomposition with OpenMP*

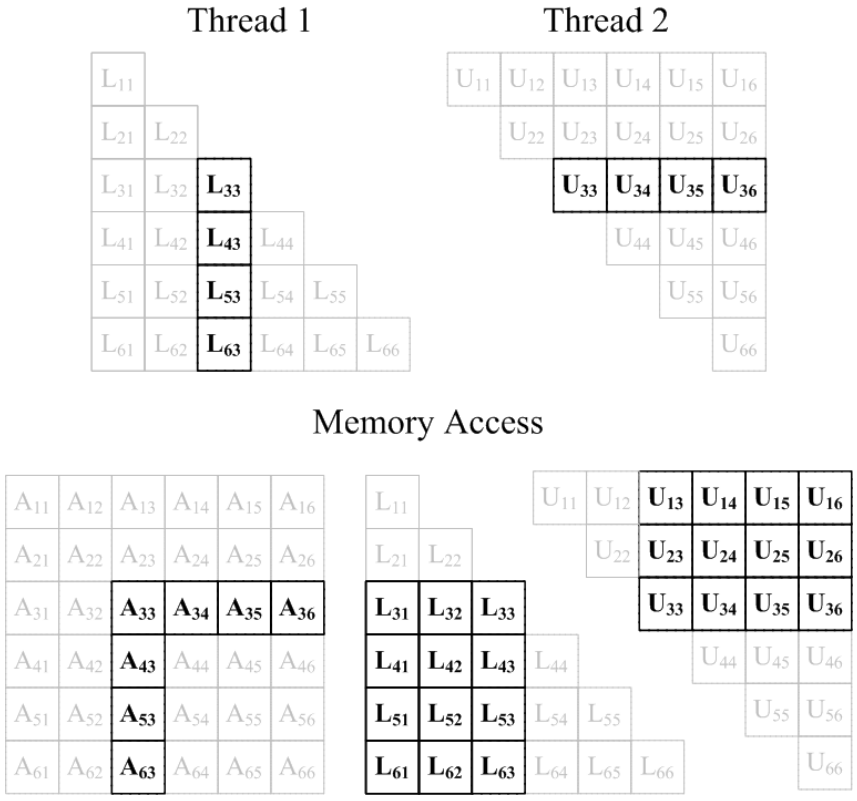


Figure 14 – *Shared memory access in parallel LU decomposition*

Decreasing costs in PC hardware has led to the use of PCs in the development of high-performance computing clusters. PC clusters are classified as distributed memory systems and consist of multiple PCs linked together by a high-speed network as shown in Figure 15. In

distributed memory systems, the programmer must specify how the nodes will distribute data across the network. This is typically done using a protocol such as MPI (message passing interface) [20]. Figure 16 shows the memory access patterns for LU decomposition in a distributed memory system.

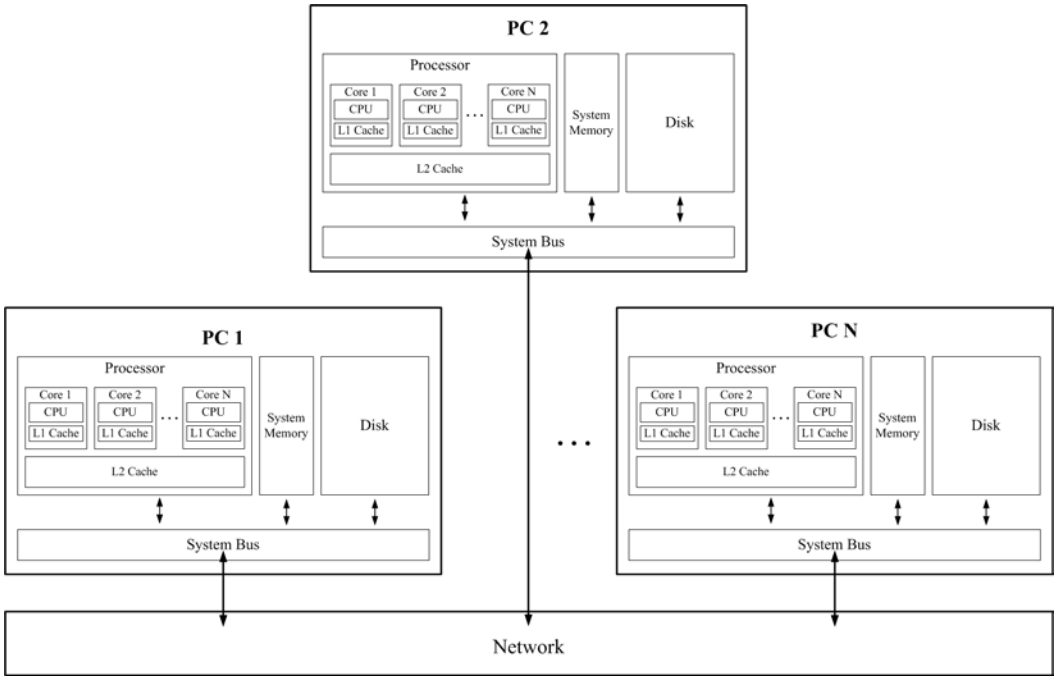


Figure 15 – *Distributed memory (PC-Cluster) architecture*

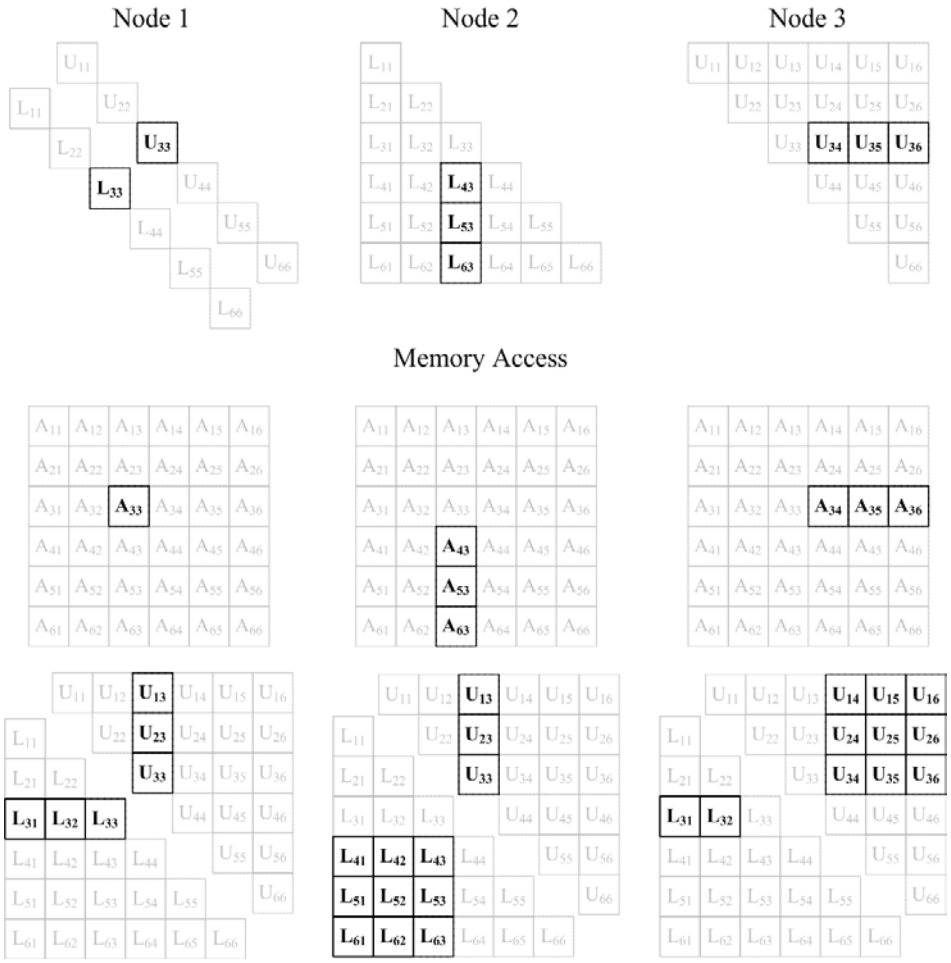


Figure 16 – *Memory access in distributed LU decomposition*

2.2 Distributed Sparse LU Decomposition

In order for a sparse matrix solver to achieve optimal performance on a distributed system, there should be special considerations for load distribution, memory usage, and communication overhead [20]. There are two steps commonly used to achieve good load distribution in sparse matrix solvers. The first step is to reorder the matrix using hypergraph partitioning as described in [14] so that matrix elements are arranged into groups. The second step is to partition the matrix into blocks in order to distribute the work among the compute nodes.

There are several data structures that can be used in sparse matrix solvers in order to achieve efficient memory usage. For example, large sparse matrices are usually stored in a compressed row or compressed column format to eliminate memory consumption of elements that are zeros. Once the matrix has been partitioned, the matrix blocks are typically stored in a dense format to improve the speed of the computation phase. The blocks, however, are organized using linked lists to eliminate the memory consumption of blocks that contain only zeros. These data structures are described in detail in Chapter III.

2.3 Numerical Stability of Distributed Sparse Solvers

Accurately solving linear systems of equations requires some form of pivoting in order to reduce the impact of rounding errors and to avoid zero pivots which would result in division by zero [2]. Linear systems that fit entirely in system memory can be solved efficiently and accurately using partial pivoting or full pivoting. Larger equations, however, require special pivoting algorithms since parts of the matrix may reside in the disk or in other machines in a network. Traditional pivoting algorithms are not used in distributed memory machines because the communication requirements would result in slow performance [13]. One state-of-the-art solver is called SuperLU [9] and is available in three versions. SuperLU partitions the matrix elements into groups called supernodes [21] for high performance. The sequential and multi-threaded versions, SuperLU and SuperLU_MT respectfully, use partial pivoting to achieve high accuracy. The distributed version however, SuperLU_DIST [9], uses a technique called static pivoting [13] that avoids pivoting during the factorization stage. The technique involves four steps. First, the matrix is scaled and large elements are permuted to the diagonal. Next, the matrix is reordered using a minimum degree algorithm in order to reduce fill-in during factorization. As shown in Figure 17, static pivoting does not always produce similar orderings

with every matrix. The matrix is then factored into lower and upper triangular matrices. Any small pivots that are found during the factorization process are replaced by $\sqrt{\varepsilon} \cdot \|A\|$ where ε is the machine precision and $\|A\|$ is the matrix norm. Finally, the system is solved using iterative refinement to correct any rounding errors.

#2 fd18, materials problem
Size: 16428x16428
Non-zeros: 63406

#10 mult_dcop_03, subsequent circuit simulation problem
Size: 25187x25187
Non-zeros: 193216

#23 twotone, frequency-domain circuit simulation problem
Size: 120750x120750
Non-zeros: 1206265

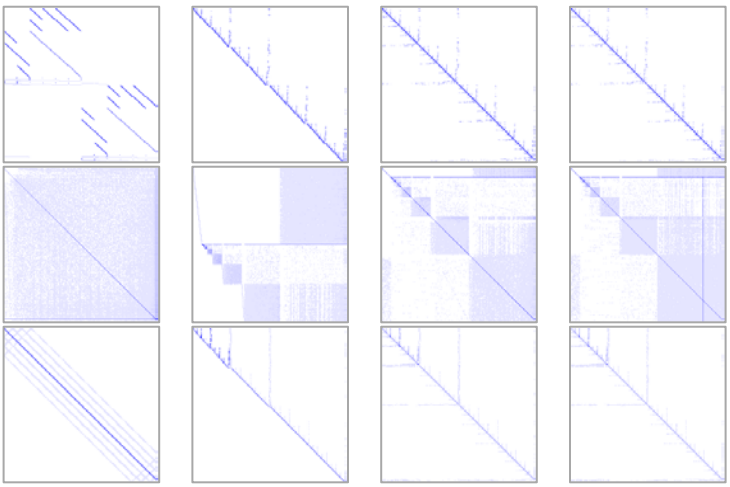


Figure 17 – *Example of static pivoting with 3 benchmark matrices*

2.4 Problem Statement and Purpose Statement

Unfortunately, static pivoting alone does not always produce accurate results due to large pivot growth [13]. Figure 18 compares the accuracy between SuperLU_MT and SuperLU_DIST. The figure shows that SuperLU_MT achieves 100% accuracy for all the benchmark matrices. SuperLU_DIST, however, does not always achieve 100% accuracy. For benchmarks 9 and 26, for example, SuperLU_DIST achieves close to 0% accuracy. The main difference between the two versions is that SuperLU_MT uses partial pivoting while SuperLU_DIST uses static pivoting. Partial pivoting leads to better accuracy because it actively controls the growth factor during Gaussian elimination [13].

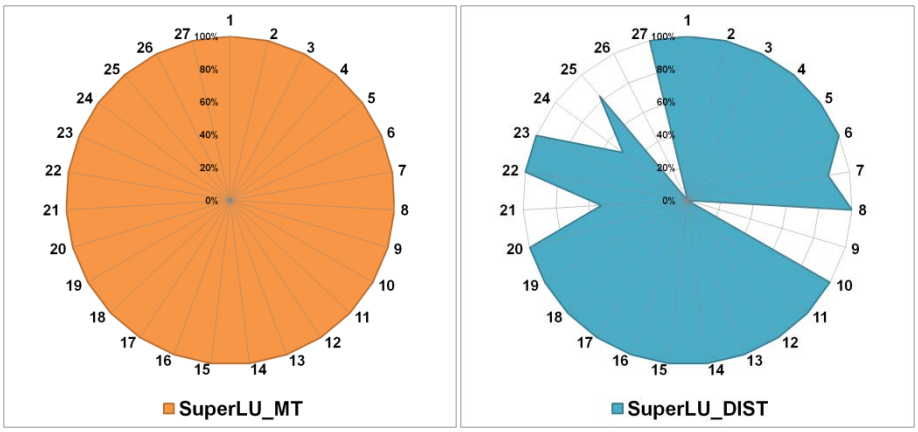


Figure 18 – *Accuracy of SuperLU_MT and SUPERLU_DIST*

The purpose of our research is to investigate pivoting algorithms and to develop a distributed sparse solver with better accuracy than current state-of-the-art distributed solvers such as SuperLU_DIST.

CHAPTER III

DISTRIBUTED SPARSE MATRIX SOLVER

We developed a sparse distributed solver called Block Sparse LU Decomposition (BSLD) in C++. In this chapter, we discuss the implementation details of our distributed solver and the pivoting algorithms that were used.

3.1 Basic Block Sparse LU Decomposition

The BSLD performs LU decomposition as described by the pseudo-code in Figure 19. The program assigns one processor to compute the factorization of the diagonal blocks. The remaining processors are divided into two groups: one group computes the factorization of the lower blocks and the other group computes the factorization of the upper blocks. The processors work on different sections of the matrix depending on the row or column index of the block and the id assigned to each processor. Before moving to the next diagonal block, all processors broadcast the factored blocks to the other processors.

Let A be a matrix of size M-by-N which is partitioned into blocks of size m-by-n.
Let N_PROCS be the number of processors available to the program.
Let NUM_PROCS_U = $\lfloor (N\_PROCS - 1)/2 \rfloor$.
Let NUM_PROCS_L = $N\_PROCS - NUM\_PROCS\_U - 1$.
Let $0 \geq PROCESSOR < N\_PROCS$.

```
for i=1 to M
         if [ PROCESSOR=0 ] then
            Factorize block A_ii into L_ii and U_ii
            Send blocks L_ii and U_ii to: 1≥PROCESSOR<N_PROCS.
            Synchronize with 1≥PROCESSOR<N_PROCS.
            Receive L blocks in column i from 1≥PROCESSOR<N_PROCS.
            Receive U blocks in row i from 1≥PROCESSOR<N_PROCS.
```

```
          else
             Receive blocks L_ii and U_ii from PROCESSOR 0.
             if [ PROCESSOR ≤ ⌊N_PROCS/2⌋ ] then
                for j=i+1 to M
                   if [ PROCESSOR=(j%NUM_PROCS_L)+1 ]
                   then
                      Factorize block L_ji
                   endif
             else
                for j=i+1 to N
                   if [ PROCESSOR=((j%NUM_PROCS_U)+NUM_PROCS_L+1)]
                   then
                      Factorize block U_ij
                   endif
             Synchronize with all processors.
             Send L blocks in column i to all processors.
             Send U blocks in row i to all processors.
endfor
```

Figure 19 – *Pseudo-code for LU decomposition with BSLD*

3.2 Sparse Matrix File Formats

At the bottom of the memory hierarchy is the slowest memory element: the disk. In order to reduce the number of read and write operations, sparse matrices are usually stored to disk in a compressed file format such as Rutherford-Boeing [22]or Matrix Market [22][23]. These formats store only the size of the matrix and its non-zeros, thereby reducing the matrix storage space and the load time. In the BSLD, we used the RBio library from the SuiteSparse library [24] to read and write Rutherford-Boeing matrix files.

3.3 Compressed Sparse Matrix Formats

At the system memory level, several different data structures are used to store matrices. In practice, permutation matrices are never stored using a dense format; instead, a permutation vector is used to represent a permutation matrix as shown in Figure 20.

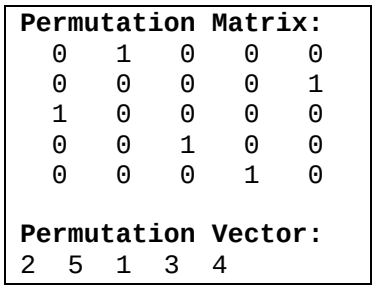


Figure 20 – *Sample permutation vector*

Because system memory is faster than the disk, parts of the matrices can be stored in a dense format such as row-major or column-major order. In order to avoid wasting system memory with many dense blocks, sub-matrices can also be stored using sparse matrix formats such as compressed-sparse-row (CSR) and compressed-sparse-column (CSC) format. CSR and CSC formats are commonly used by many sparse matrix libraries such as SuiteSparse [24]. These formats can also be used to efficiently transport sub-matrices in a distributed memory system. Figure 21 shows a sample matrix stored in CSR and CSC format [25]. In the CSR format, non-zero elements are stored using three arrays. One array holds all the non-zero values of the matrix. The col. index array holds the column index for each of the non-zero elements of the matrix. The row pointer array holds the index of the first element in each row of the matrix. The last element in the row pointer array holds the total number of non-zeros in the matrix. The CSC format is similar to the CSR format. In the CSC format, however, the non-zero values are organized by column instead of by rows. The row index array holds the row index for non-zero element in the matrix and the col. pointer array holds the index of the first element in each column of the matrix. The last element in the col. pointer array holds the total number of non-zero elements in the matrix.

```
Matrix:
  2   0   7   0   0
  0   0   0   6   0
  0   5   0   0   1
  4   0   0   7   0
  0   0   0   6   2

CSR Format:
      Value:  2  7  6  5  1  4  7  6  2
  Col Index:  1  3  4  2  5  1  4  4  5
Row Pointer:  1  3  4  6  8  9

CSC Format:
      Value:  2  4  5  7  6  7  6  1  2
  Row Index:  1  4  3  1  2  4  5  3  5
Col Pointer:  1  3  4  5  8  9
```

Figure 21 – *Sample matrix in CSR and CSC format*

3.4 Block Matrix Data Structures

Sparse matrices are usually stored using several levels of organization within system memory. At the lower level, blocks or nodes are stored using dense or compressed formats as described above. At the higher level, these blocks are organized using data structures such as linked lists. As shown in Figure 22, the organization of the blocks at the higher level can help predict where non-zero blocks may appear later during factorization.

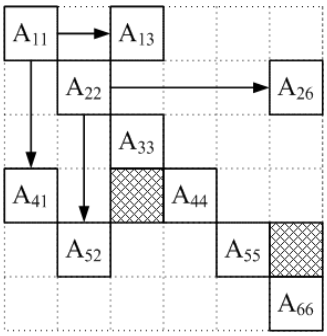


Figure 22 – *Block matrix organization*

3.5 Basic Linear Algebra Subroutines (BLAS)

In practice, matrix operations can be computed much faster using small dense sub-matrices instead of compressed matrix formats thanks to finely tuned math libraries such as the

Basic Linear Algebra Subroutines (BLAS) [26]. These high performance math libraries have been tuned to exploit spatial and temporal locality and are used extensively in scientific and mathematical software. In the BSLD, we used the BLAS to improve the performance of LU decomposition.

3.6 Pivoting Algorithms

The BSLD uses several different pivoting algorithms to help improve the accuracy of the final solution. Figure 23a shows a sample matrix, Zhao2 [27], which will be used to illustrate static pivoting [13] as follows:

- Step 1: Reordering the sparse matrix using hypergraph-based nested dissection as described in [14]. In this step, we use the PaToH [28] library to perform hypergraph partitioning. Our solver allows different parameters to be specified for the number of partitions per dissection and the maximum size of each partition. Figure 23b shows a sample matrix, Zhao2 [27], after hypergraph-based nested dissection. Nested dissection re-orders the matrix so that the matrix elements are arranged into groups. Compared to the original matrix in Figure 23a, the reordered matrix in Figure 23breduces fill-in and allows better load distribution; and
- Step 2: Permuting large elements to the diagonal and scaling the matrix. In this step, our solver uses the MC64 routine as described in [13]. This routine performs weighted matching on the bipartite graph of the matrix in order to permute the large elements to the diagonal. It then scales the matrix so that the diagonal elements are equal to one. Figure 23c shows the sample matrix after applying the

MC64 routine. Compared to the matrix in Figure 23b, the reordered matrix in Figure 23c helps reduce rounding errors by strengthening the diagonal; and

- Step 3: Using a minimum degree algorithm to reduce fill-in during factorization. In our solver, we use the constrained minimum degree (CCOLAMD) routine [17] as described in [13]. This routine applies a column approximate minimum degree ordering to the different sections defined by the hypergraph partitions from Step 1. Figure 23d shows the sample matrix after the CCOLAMD routine. Compared to the matrix in Figure 23c, the reordered matrix in Figure 23d helps reduce fill-in even further.

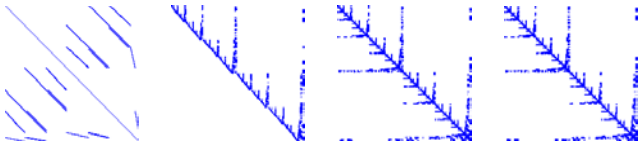


Figure 23 – *Matrix pre-conditioning*

During the LU decomposition phase, the BSLD can perform partial pivoting or threshold pivoting [7]. In this thesis, we focus more on developing good accuracy in our distributed solver by comparing these different pivoting strategies.

3.7 Proposed Block Sparse LU Decomposition

The proposed solver uses block sparse LU decomposition with static pivoting and threshold pivoting as shown in Figure 24.The threshold pivoting algorithm swaps rows only if the pivot element is less than the perturbation value which is the product of the machine accuracy and the matrix norm.

```
1. Static Pivoting
2. Matrix Partitioning
3. Threshold Pivoting with Perturbation
   as follows:
   if |U_ii| <= √ε · ‖A‖ then
      Compute U_ii from un-factored rows j to m
      let U_ij = max(U_ii)
      if U_ij < √ε · ‖A‖
         U_ii = √ε · ‖A‖
      else
         Swap rows i and j
      endif
   endif
4. Back Substitution with Iterative Refinement
```

Figure 24 –*Proposed block sparse LU decomposition*

The main difference between the proposed BSLD and SuperLU_DIST is that the BSLD uses square blocks for matrix partitioning and threshold pivoting within blocks during the factorization stage.

CHAPTER IV

SIMULATION METHODOLOGY AND RESULTS

4.1 Sparse Matrix Benchmarks

The BSLD was tested with 27 benchmark matrices from the University of Florida sparse matrix collection [27]. These matrices were used in [14] to check for fill-in during LU factorization. Table 1 lists our set of test matrices, their name, discipline, number of non-zeros (NNZ), and sparsity.

#	Name	Discipline	Size	Order	NNZ	Sparsity (%)
5	swang1	semiconductor device problem sequence	small	3169	20841	99.79
8	raefsky6	structural problem	small	3402	130371	98.87
1	lns_3937	computational fluid dynamics problem	small	3937	25407	99.84
7	lhr04	chemical process simulation problem	small	4101	81057	99.52
14	sinc12	materials problem	small	7500	283992	99.50
6	mark3jac020	economic problem	small	9129	52883	99.94
18	sinc15	materials problem	small	11532	551184	99.59
3	poli_large	economic problem	small	15575	33033	99.99
2	fd18	materials problem	small	16428	63406	99.98
22	sinc18	materials problem	medium	16428	948696	99.65
19	e40r0100	2D/3D problem	medium	17281	553562	99.81
11	shermanACb	2D/3D problem	medium	18510	145149	99.96
4	bayer04	chemical process simulation problem	medium	20545	85537	99.98
20	Zd_Jac2_db	chemical process simulation problem	medium	22835	676439	99.87
17	af23560	computational fluid dynamics problem	medium	23560	460598	99.92
10	mult_dcop_03	subsequent circuit simulation problem	medium	25187	193216	99.97
9	Zhao2	electromagnetics problem	medium	33861	166453	99.99
21	lhr34c	chemical process simulation problem	medium	35152	764014	99.94
16	onetone1	frequency-domain circuit simulation problem	large	36057	335552	99.97
27	bbmat	computational fluid dynamics problem	large	38744	1771722	99.88
26	av41092	2D/3D problem	large	41092	1683902	99.90
12	jan99jac120sc	economic problem	large	41374	229385	99.99
13	bayer01	chemical process simulation problem	large	57735	275094	99.99
15	mark3jac140sc	economic problem	large	64089	376395	99.99
24	lhr71c	chemical process simulation problem	large	70304	1528092	99.97
25	torso2	2D/3D problem	large	115967	1033473	99.99
23	twotone	frequency-domain circuit simulation problem	large	120750	1206265	99.99

Table 1 – *The set of benchmark matrices*

4.2 Simulation Environment

Our simulation environment consisted of a multi-core PC and a PC cluster running a Linux operating system. The multi-core PC consisted of an Intel Xeon X5550 running at 2.66 GHz with 8 MB of Smart Cache, 4 Cores capable of running 8 threads total and 20 GB of system memory. The PC cluster consisted of 72 nodes, each with 2 Intel Xeon X5650 processors running at 2.66 GHz. Each processor had 12 MB of Smart Cache with 6 cores and was capable of running 12 threads. In addition, each node had 48 GB of RAM.

4.3 Simulation Procedure

To test the BSLD, we first re-ordered each of the 27 benchmark matrices using the static pivoting algorithms described in Chapter III. The parameters used for hypergraph-based nested dissection were 2 partitions per dissection with a maximum partition size of 8 cells. We then multiplied each coefficient matrix A by x where $x_i=i$ in order to get the vector of constants b. This would allow us to set theoretical values for the equation $Ax=b$. Finally, we solved $Ax=b$ using SuperLU_DIST and BSLD. For each of the matrices tested, we computed the accuracy by counting the number of experimental solutions, x_i, where $|x_i - i| < 0.2$ and dividing by the number of variables. We chose an arbitrary threshold of 0.2 in order to establish a consistent measure of accuracy for all the benchmarks.

4.4 Execution Times

The execution time for each benchmark matrix was tested using different number of cores on the multi-core system and different number of processors on the PC cluster.

Table 2 shows the execution times for each of the benchmark matrices using 3, 5, 7, and 9 processors on the distributed memory system. The results show that for the larger matrices, the execution time decreases as the number of processors increases. For example, for benchmark 27-

bbmat in Table 2, the execution time using 9 processors (780.91 seconds) was much faster than the time using 3 processors (1199.27 seconds). For smaller matrices, however, execution time begins to increase as the number of processors increases. For example, for benchmark 03-poli_large in Table 2, the execution time with 9 processors (0.41 seconds) is slower than the execution time with 3 processors (0.29 seconds).

Matrix	Size	# of Processors			
		3	5	7	9
05-swang1	small	0.91	0.87	0.9	0.93
08-raefsky6	small	1.68	1.44	1.44	1.52
01-lns_3937	small	2.61	2.16	2.18	2.22
07-lhr04	small	2.17	1.89	1.92	1.99
14-sinc12	small	279.26	207.31	186.87	178.34
06-mark3jac020	small	19.09	15.06	14.06	13.93
18-sinc15	small	620.63	417.61	375.21	350.23
03-poli_large	small	0.29	0.31	0.36	0.41
02-fd18	small	8.15	7.06	7.19	7.33
22-sinc18	medium	1654.96	1226.36	1083.13	1011.76
19-e40r0100	medium	39.49	29.6	27.01	26.25
11-shermanACb	medium	170.09	103.23	78.19	71.18
04-bayer04	medium	2.85	2.71	2.88	3.1
20-Zd_Jac2_db	medium	27.75	20.33	18.38	17.94
17-af23560	medium	148.96	103.05	90.05	84.07
10-mult_dcop_03	medium	107.81	51.09	44.96	44.19
09-Zhao2	medium	185.77	129.29	116.67	111.08
21-lhr34c	medium	19.43	16.79	16.93	17.52
16-onetone1	large	65.69	49.72	45.09	43.43
27-bbmat	large	1199.27	914.31	824.11	780.91
26-av41092	large	621.86	390.81	341.14	315.68
12-jan99jac120sc	large	34.47	21.15	19.28	19.02
13-bayer01	large	7.44	7.19	7.72	8.37
15-mark3jac140sc	large	299.6	222.43	200.76	191.51
24-lhr71c	large	37.95	34.46	34.53	35.61
25-torso2	large	97.45	78.63	75.26	75.62
23-twotone	large	857.66	576.09	479.65	437.26

Table 2 – *Execution times on the PC cluster*

Table 3 shows the execution times for each of the benchmark matrices using 3, 4, 5, and 6 cores on the shared memory system. The results show that the shared memory system was about twice as fast as the distributed memory system. For example, for benchmark 05-swang1, the execution time on the multi-core PC with 3 cores (0.44 seconds) was faster than the execution time on the PC cluster with 3 processors (0.91 seconds). One reason for the difference

in results is that the distributed memory system introduces some delays due to inter-node communication, as shown in [4]. One way to improve the results in the distributed memory system would be modify the scheduler affinity to make better use of the multi-core processors as described in [4].

Matrix	Size	# of Cores			
		3	4	5	6
05-swang1	small	0.44	0.46	0.42	0.47
08-raefsky6	small	0.83	0.85	0.74	0.81
01-lns_3937	small	1.21	1.27	1.04	1.15
07-lhr04	small	1.00	1.03	0.95	1.02
14-sinc12	small	164.35	163.51	137.83	140.28
06-mark3jac020	small	11.02	11.43	9.08	9.57
18-sinc15	small	331.54	334.02	268.92	268.79
03-poli_large	small	0.18	0.20	0.21	0.24
02-fd18	small	4.58	4.87	4.05	4.39
22-sinc18	medium	990.68	991.73	811.31	818.98
19-e40r0100	medium	18.95	20.25	14.45	15.78
11-shermanACb	medium	89.32	91.04	59.83	62.94
04-bayer04	medium	1.57	1.67	1.54	1.65
20-Zd_Jac2_db	medium	12.67	13.62	10.14	10.76
17-af23560	medium	73.08	73.63	52.35	55.22
10-mult_dcop_03	medium	65.92	68.56	44.73	47.74
09-Zhao2	medium	106.05	116.18	85.73	88.39
21-lhr34c	medium	9.66	10.10	9.12	9.73
16-onetone1	large	43.90	41.93	33.47	34.34
27-bbmat	large	724.85	733.50	618.48	615.76
26-av41092	large	343.52	356.56	262.78	268.36
12-jan99jac120sc	large	16.26	19.24	13.18	14.10
13-bayer01	large	4.30	4.58	4.28	4.72
15-mark3jac140sc	large	191.67	197.75	146.58	154.42
24-lhr71c	large	20.72	21.17	18.89	20.14
25-torso2	large	55.93	54.13	43.68	46.46
23-twotone	large	568.61	574.82	407.29	412.29

Table 3 – *Execution times on the shared memory system*

4.5 Accuracy Measurements

The accuracy for each pivoting algorithm was measured using the 27 benchmark matrices. Our focus was to find a pivoting algorithm that will produce the best accuracy for distributed LU decomposition. The results show that for 20 matrices, all solvers were able to obtain 100% accuracy.Table 4 shows the results for the benchmarks that were not able to achieve 100% accuracy with SuperLU_DIST. The table shows that for larger matrices, our threshold

pivoting algorithm achieved the most benefit. For example, benchmark 21-lhr34 showed 47.54% improvement in accuracy, 24-lhr71 showed 51.16% improvement, and 25-torso2 showed 16.91% improvement. Figure 25 shows that for the benchmarks where SuperLU_DIST achieved very low accuracy, the BSLD was not able to improve the results by very much.

Benchmark	Size	SuperLU_DIST	BSLD		Average Improvement (%)
			Partial Pivoting	Threshold Pivoting	
07-lhr04	small	86.66%	100.00%	100.00%	13.34%
10-mult_dcop_03	medium	99.68%	99.70%	99.70%	0.02%
09-Zhao2	medium	0.01%	0.94%	0.14%	0.53%
21-lhr34c	medium	52.41%	99.84%	99.95%	47.49%
26-av41092	large	0.01%	21.84%	0.02%	10.92%
24-lhr71c	large	48.80%	80.75%	99.96%	41.56%
25-torso2	large	83.09%	100.00%	100.00%	16.91%

Table 4 – *Accuracy for 7 benchmark matrices*

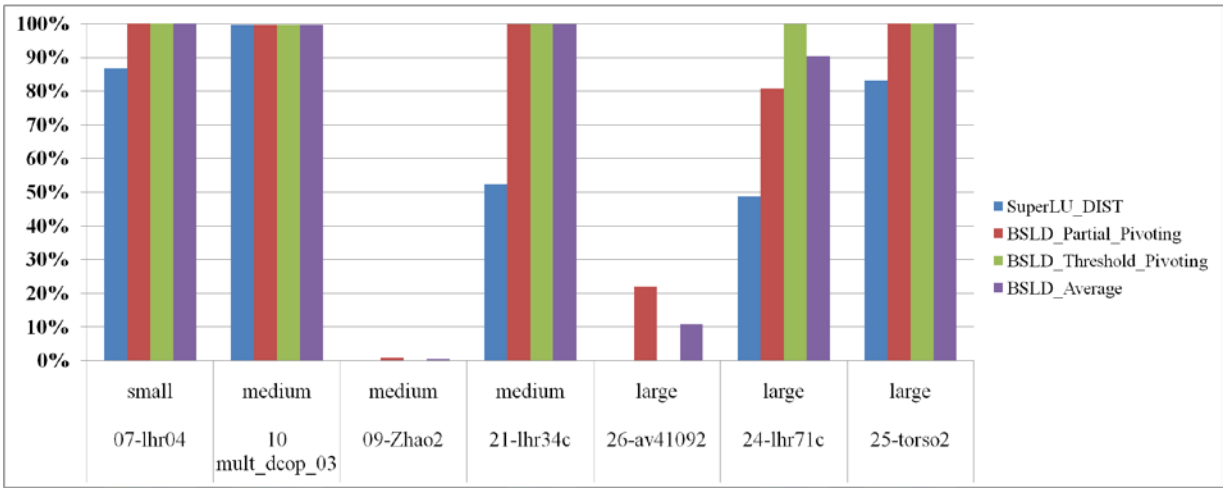


Figure 25 –*Accuracy for 7 benchmark matrices*

Next, we observed how accuracy was affected by block size in the BSLD. Figure 26shows the results for the benchmarks that were not able to achieve 100% accuracy with SuperLU_DIST. In the two cases where the BSLD did not show much improvement, the results show that the larger block sizes had a better impact on the accuracy than the smaller block sizes.

For example, in Figure 28, benchmark 9 had an accuracy of about .90% with block size 32 and .49% with block size 256 while the accuracy for other block sizes was below .2%. Benchmark 10 had an accuracy of 99.7% with block size 256 while the accuracy with the other block sizes remained at 99.68%. In Figure 29, benchmark 26 had an accuracy of 21.84% with block size 256 while the accuracy with other block sizes was less than 2%.In most cases, however, threshold pivoting with large block sizes provided the best improvement in accuracy. For example, in Figure 27, the best accuracy was achieved with block size 256.

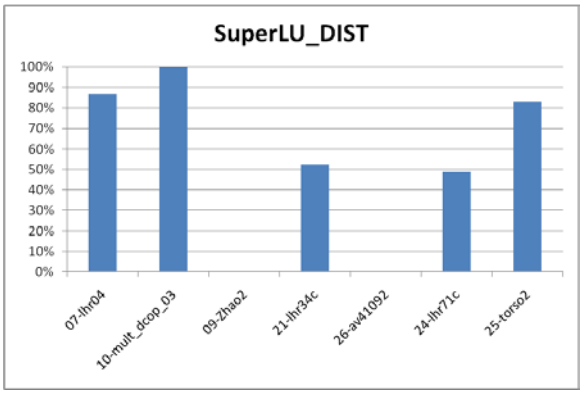


Figure 26 –*Accuracy of SuperLU_DIST for 7 benchmarks*

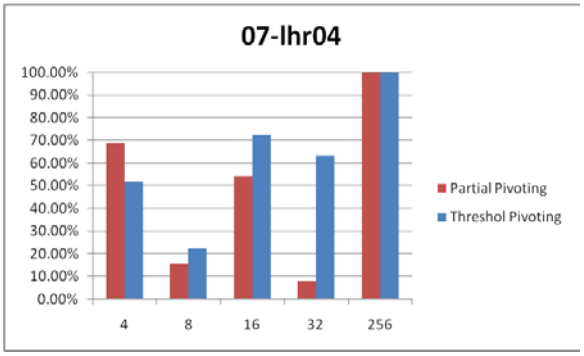


Figure 27 –*Accuracy of BSLD for small benchmark 07-lhr04*

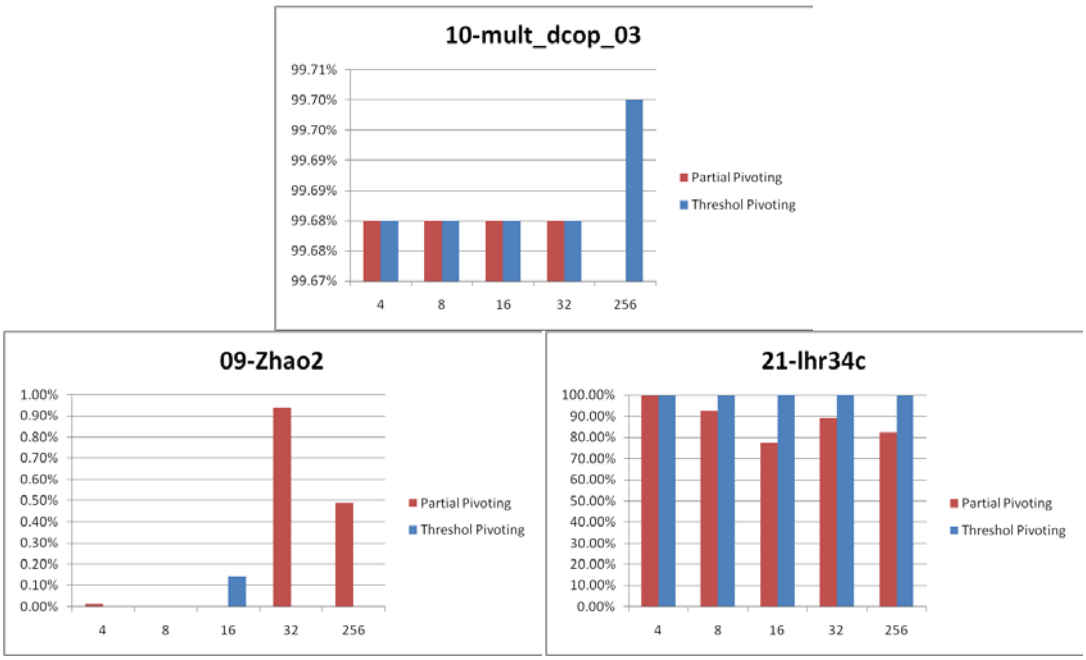


Figure 28 *–Accuracy of BSLD for medium benchmarks*

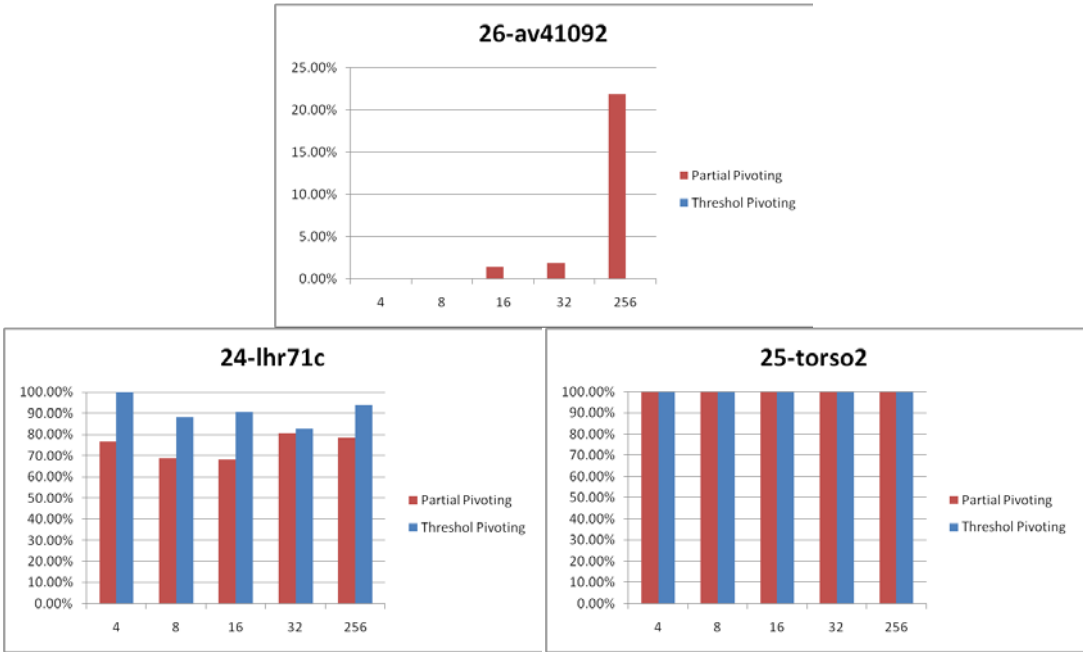


Figure 29 *–Accuracy of BSLD for large benchmarks*

CHAPTER V

SUMMARY AND CONCLUSION

5.1 Summary and Conclusion

The increasing complexity of modern scientific and engineering problems has created a need to solve large, sparse, linear equations. To solve these equations efficiently, modern computing systems have shifted to parallel and distributed systems. The algorithms used to solve large equations on these systems must take special care to avoid sacrificing numerical stability in order to reduce the computation time. We have presented a distributed sparse matrix solver that takes these issues into account and yields better accuracy than a current, state-of-the-art, distributed solver. Our main contribution is a threshold pivoting algorithm that uses square blocks to achieve good accuracy in distributed memory systems. Our results show that our block based threshold pivoting algorithm improves accuracy for large linear equations by up to 51.16% compared to static pivoting. We are not aware of any distributed solvers that use the techniques as we have described.

5.2 Future Work

In order to further improve the accuracy of our solver, it may help to investigate the accuracy while using rectangular blocks for the matrix partitioning. In our solver, we used square blocks to simplify the distribution of work in the factorization process. However, we can increase the number of elements that are considered for pivot selection by increasing the number of rows in the blocks and we can reduce memory consumption and floating point operations by reducing

the number of columns in the blocks. Therefore, rectangular blocks can increase accuracy and improve execution time compared with square blocks. However, rectangular partitioning would also introduce some complexity in work distribution since there would be more blocks per row than per column. In addition, rectangular blocks would have to be arranged as shown in Figure 30bin order to allow pivoting. Pivoting would not be possible in Figure 30a because pivots must be selected from elements below the diagonal. This would also affect work distribution since it would increase data dependencies between blocks. An elimination graph may help alleviate these issues and may also further improve the execution times.

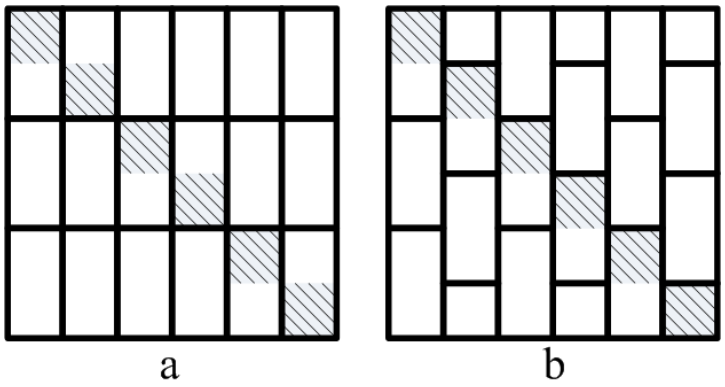


Figure 30 –*Rectangular block partitioning*

The speed may also be improved by experimenting with different BLAS libraries such as a sparse BLAS implementation. This would allow us to use bigger blocks using compressed matrix formats and optimized sparse matrix operations to reduce execution times. We are currently working to improve the performance of the BSLD by optimizing the scheduler to make better use of the multi-core PC cluster as described in [4].

REFERENCES

[1] Lay, David C., Linear Algebra and its Applications, Addison Wesley, 2003.

[2] Demmel, James W. Applied Numerical Linear Algebra. Philadelphia: SIAM, 1997.

[3] Shalf, J., "The New Landscape of Parallel Computer Architecture", J. Phys.: Conf. Ser. 78 012066

[4] Chai L., Gao Q., Panda D. K., "Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System", Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on, pp. 471-478

[5] Alexander, Charles K., and Matthew N. O. Sadiku. Fundamentals of Electric Circuits. 3rd ed. New York: McGraw-Hill, 2007.

[6] Cadence Design Systems, Inc., "Cadence PSpice A/D and Advanced Analysis." <http://www.cadence.com/products/orcad/pspice_simulation>

[7] Trefethen, Lloyd N., and David Bau, III, Numerical Linear Algebra. Philadelphia: SIAM, 1997.

[8] Saad, Yousef; Iterative Methods for Sparse Linear Systems, 2nd Edition, SIAM, 2003.

[9] X.S. Li, J. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems", presented at ACM Trans. Math. Softw., 2003, pp.110-140.

[10] O. Schenk, K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO", Future Generation Computer Systems, ISSN 0167-739X, 2004, Volume 20, Issue 3, pp. 475-487

[11] A. Gupta, "A Shared- and distributed-memory parallel general sparse direct solver", Applicable Algebra in Engineering, Communication and Computing, ISSN 0938-1279, 05/2007, Volume 18, Issue 3, pp. 263-277

[12] Hwang, K, and Cheng, Y.H., "Partitioned Matrix Algorithms for VLSI Arithmetic Systems", IEEE Transactions on Computers, Vol. C-3 1, no. 12, December 1982

[13] X. S. Li, J. W. Demmel, "Making Sparse Gaussian Elimination Scalable by Static Pivoting", In: SC: IEEE, 1998, S. 34

[14] L. Grigori, E.G. Boman, S. Donfack, and T.A. Davis, "Hypergraph-Based Unsymmetric Nested Dissection Ordering for Sparse LU Factorization", presented at SIAM J. Scientific Computing, 2010, pp.3426-3446.

[15] Markowitz, Harry M. "The elimination form of the inverse and its application to linear programming." Management Science vol. 3 no. 3. 1957. pp. 255-269.

[16] P. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm", SIAM Journal on Matrix Analysis and Applications, vol 17, no. 4, pp. 886-905, Dec. 1996.

[17] T. A. Davis, J. R. Gilbert, S. Larimore, E. Ng. "COLAMD, CCOLAMD: A column approximate minimum degree ordering algorithm", presented at ACM Trans. Math. Softw., 2004, pp. 353-376.

[18] J. A. Scott, "Scaling and Pivoting in an Out-of-Core Sparse Direct Solver", ACM Transactions on Mathematical Software, ISSN 0098-3500, 2011, Volume 37, Issue 2, p. 19

[19] Tian, Tian and Shih, Chiu-Pi, "Software Techniques for Shared-Cache Multi-Core Systems." Intel Corporation, 2012 <http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems>

[20] Lin, Calvin and Snyder, Lawrence, Principles of Parallel Programming. Addison-Wesley, 2008.

[21] Demmel, James W., et al."A supernodal approach to sparse partial pivoting", SIAM J. Matrix Analysis and Applications, 1999, vol. 20 no. 3 pp. 720-755

[22] I.S. Duff, R.G. Grimes, and J.G. Lewis, "The Rutherford-Boeing Sparse Matrix Collection," Report RAL-TR-97-031, Revision 1, 1999

[23] National Institute of Standards and Technology, “Matrix Market Text File Formats”, <http://math.nist.gov/MatrixMarket/formats.html>

[24] T. Davis; "SuiteSparse", <http://www.cise.ufl.edu/research/sparse/SuiteSparse/>

[25] R. Barrett, M. Berry, T. F. Chan, et. al, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, Philadelphia: SIAM, 1994. <http://netlib.org/linalg/html_templates/report.html>

[26] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Soft., 16 (1990), pp. 1--17.<http://www.netlib.org/blas/>

[27] T. Davis. University of Florida Sparse Matrix Collection. NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997.

[28] U. V. Catalyurek and C. Aykanat. "Patoh: Partitioning tool for hypergraphs." User's Guide, 1999

APPENDIX

APPENDIX

SOURCE CODE LISTING

```
    1: /*
    2:     Author: Esteban Torres
    3:     Date: August 2011
    4:     File: blockmatrix.cpp
    5:     Purpose: This file contains the implementation details for the block
matrix
    6:              data structure.
    7: */
    8: #include "blockmatrix.h"
    9: #include "datastructures.h"
   10: #include <cmath>
   11:
   12:
   13: //matrixNode-----------------------------------------------------------
   14: matrixNode::matrixNode(int bsz):
   15:     blockSize(bsz),
   16:     row(0),
   17:     col(0),
   18:     rightNode(NULL),
   19:     bottomNode(NULL),
   20:     sizeM(bsz),
   21:     sizeN(bsz)
   22: {
   23:     block = new CSFBlock(bsz,bsz);
   24:     assert(block!=NULL);
   25: }
   26: matrixNode::matrixNode( const matrixNode &B ):
   27:     blockSize(B.blockSize),
   28:     row(B.row),
   29:     col(B.col),
   30:     rightNode(NULL),
   31:     bottomNode(NULL),
   32:     sizeM(B.sizeM),
   33:     sizeN(B.sizeN)
   34: {
   35:     block = new CSFBlock( B.blockSize, B.blockSize );
   36: }
   37: matrixNode::matrixNode(int bsz,int r,int c):
   38:     blockSize(bsz),
   39:     row(r),
   40:     col(c),
   41:     rightNode(NULL),
   42:     bottomNode(NULL),
   43:     sizeM(0),
   44:     sizeN(0)
   45: {
   46:     block = new CSFBlock(bsz,bsz);
   47: }
   48: void matrixNode::operator= ( const matrixNode *B )
   49: {
   50:     *block = *(B->block);
   51: }
   52: matrixNode* matrixNode::toCSR()
   53: {
   54:     matrixNode *csrNode = new matrixNode(blockSize, row, col);
   55:     csrNode->sizeM = sizeM;
   56:     csrNode->sizeN = sizeN;
   57:     *(csrNode->block) = block->toCSR();
   58:     return csrNode;
   59: }
   60: void matrixNode::reserve( int size )
   61: {
   62:     block->reserve( size );
```

```
   63: }
   64: void matrixNode::setIndex(int r, int c)
   65: {
   66:     row = r;
   67:     col = c;
   68: }
   69: void matrixNode::insertValueCSC(int r, int c, double value)
   70: {
   71:     if( value==0 )
   72:         return;
   73:     block->insertCSC(r,c,value);
   74: }
   75: void matrixNode::insertValueCSR(int r, int c, double value)
   76: {
   77:     if( value==0 )
   78:         return;
   79:     block->insertCSR(r,c,value);
   80: }
   81: bool matrixNode::isEnd()
   82: {
   83:     return block->isEnd();
   84: }
   85: double matrixNode::atCSR(int row, int col)
   86: {
   87:     return block->atCSR(row,col);
   88: }
   89: double matrixNode::atCSC(int row, int col)
   90: {
   91:     return block->atCSC(row,col);
   92: }
   93: void matrixNode::getFirstCSC( int &r, int &c, double &v )
   94: {
   95:     block->getFirstCSC( r, c, v );
   96: }
   97: void matrixNode::getFirstCSR( int &r, int &c, double &v )
   98: {
   99:     block->getFirstCSR( r, c, v );
  100: }
  101: bool matrixNode::getNextCSC( int &r, int &c, double &v )
  102: {
  103:     return block->getNextCSC( r, c, v );
  104: }
  105: bool matrixNode::getNextCSR( int &r, int &c, double &v )
  106: {
  107:     return block->getNextCSR( r, c, v );
  108: }
  109: void matrixNode::getCurrent( int &r, int &c, double &v )
  110: {
  111:     block->getCurrent( r, c, v );
  112: }
  113: double matrixNode::getDiagonal( int i )
  114: {
  115:     assert( row==col );
  116:     return block->getDiagonal(i);
  117: }
  118: void matrixNode::dump() const
  119: {
  120:     printf("  Block[%d][%d]:\n",row, col);
  121:     //printf("    sizeM: %d\n",sizeM);
  122:     //printf("    sizeN: %d\n",sizeN);
  123:     printf("    rightNode: "); if(rightNode==NULL) printf("NULL\n"); else
printf("[%d][%d]\n",rightNode->row,rightNode->col);
  124:     printf("    bottomNode: "); if(bottomNode==NULL) printf("NULL\n"); else
```

```
printf("[%d][%d]\n",bottomNode->row,bottomNode->col);
  125:     block->dump();
  126: }
  127: void matrixNode::printCSC() const
  128: {
  129:     printf("Block[%d][%d]:\n",row, col);
  130:     block->printCSC();
  131: }
  132: void matrixNode::printCSR() const
  133: {
  134:     printf("Block[%d][%d]:\n",row, col);
  135:     block->printCSR();
  136: }
  137: int matrixNode::nnz()
  138: {
  139:     return block->nnz();
  140: }
  141: //blockMatrix-------------------------------------------------------
  142: blockMatrix::blockMatrix(int bsz, int mm, int nn):
  143:     m(mm),
  144:     n(nn),
  145:     bm( static_cast<int>(ceil(mm/static_cast<double>(bsz))) ),
  146:     bn( static_cast<int>(ceil(nn/static_cast<double>(bsz))) ),
  147:     blockSize(bsz),
  148:     currentNode(NULL),
  149:     /*diagonals( static_cast<int>(ceil(mm/static_cast<double>(bsz))),
matrixNode(bsz) ),
  150:     leftMost(  static_cast<int>(ceil(mm/static_cast<double>(bsz))), NULL ),
  151:     topMost(   static_cast<int>(ceil(mm/static_cast<double>(bsz))), NULL )*/
  152:     diagonals( new matrixNode*[bm] ),
  153:     leftMost(  new matrixNode*[bm] ),
  154:     topMost(   new matrixNode*[bn] )
  155: {
  156:     for( int i=0; i<bn; i++ )
  157:         topMost[i] = NULL;
  158:     for( int i=0; i<bm; i++ )
  159:     {
  160:         diagonals[i] = new matrixNode(bsz);
  161:         leftMost[i]  = NULL;
  162:         diagonals[i]->reserve( bsz << 1 ); //bsz*2
  163:         diagonals[i]->setIndex(i,i);
  164:     }
  165:     if( (bm-1)*bsz+bsz > m )
  166:     {
  167:         diagonals[bm-1]->sizeM = m%bsz;
  168:         diagonals[bm-1]->sizeN = m%bsz;
  169:     }
  170: }
  171: blockMatrix::blockMatrix( blockMatrix &LM ):
  172:     m(LM.m),
  173:     n(LM.n),
  174:     bm(LM.bm),
  175:     bn(LM.bn),
  176:     blockSize(LM.blockSize),
  177:     currentNode(NULL),
  178:     diagonals( new matrixNode *[LM.bm] ),
  179:     leftMost(  new matrixNode *[LM.bm] ),
  180:     topMost(   new matrixNode *[LM.bn] )
  181: {
  182:     for( int i=0; i<bn; i++ )
  183:         topMost[i] = NULL;
  184:     for( int i=0; i<bm; i++ )
  185:     {
```

```
  186:         diagonals[i] = new matrixNode(LM.blockSize);
  187:         leftMost[i]  = NULL;
  188:         diagonals[i]->reserve( blockSize << 1 ); //bsz*2
  189:         diagonals[i]->setIndex(i,i);
  190:     }
  191:     if( (bm-1)*blockSize+blockSize > m )
  192:     {
  193:         diagonals[bm-1]->sizeM = m%blockSize;
  194:         diagonals[bm-1]->sizeN = m%blockSize;
  195:     }
  196: }
  197: blockMatrix::~blockMatrix()
  198: {
  199:     matrixNode* temp;
  200:
  201:     for( int i=0; i<bm; i++ )
  202:     {
  203:         currentNode = diagonals[i];
  204:         while( currentNode->rightNode !=NULL )
  205:         {
  206:             temp = currentNode->rightNode;
  207:             currentNode->rightNode = temp->rightNode;
  208:             delete temp;
  209:         }
  210:         while( currentNode->bottomNode !=NULL )
  211:         {
  212:             temp = currentNode->bottomNode;
  213:             currentNode->bottomNode = temp->bottomNode;
  214:             delete temp;
  215:         }
  216:     }
  217:
  218:     for( int i=0; i<bm; i++ )
  219:         delete diagonals[i];
  220:
  221:     delete [] leftMost;
  222:     delete [] topMost;
  223:     delete [] diagonals;
  224:
  225:
  226:     currentNode = NULL;
  227: }
  228: blockMatrix* blockMatrix::toCSR()
  229: {
  230:     matrixNode *temp=NULL;
  231:     blockMatrix *csrMatrix = new blockMatrix( *this );
  232:     for( int i=0; i<bm; i++ )
  233:     {
  234:         csrMatrix->diagonals[ i ] = diagonals[i]->toCSR();
  235:
  236:         csrMatrix->currentNode = csrMatrix->diagonals[i];
  237:         currentNode = diagonals[ i ]->rightNode;
  238:
  239:         while( currentNode!= NULL )
  240:         {
  241:             csrMatrix->currentNode->rightNode = currentNode->toCSR();
  242:             csrMatrix->currentNode = csrMatrix->currentNode->rightNode;
  243:             if( csrMatrix->topMost[ currentNode->col ]==NULL )
  244:                 csrMatrix->topMost[ currentNode->col ] =
csrMatrix->currentNode;
  245:             else
  246:             {
  247:                 temp = csrMatrix->topMost[ currentNode->col ];
```

```
  248:                 while( temp->bottomNode!=NULL && temp->bottomNode->row <
currentNode->row )
  249:                     temp = temp->bottomNode;
  250:                 csrMatrix->currentNode->bottomNode = temp->bottomNode;
  251:                 temp->bottomNode = csrMatrix->currentNode;
  252:             }
  253:             currentNode=currentNode->rightNode;
  254:         }
  255:
  256:         csrMatrix->currentNode = csrMatrix->diagonals[i];
  257:         currentNode = diagonals[ i ]->bottomNode;
  258:         while( currentNode!= NULL )
  259:         {
  260:             csrMatrix->currentNode->bottomNode = currentNode->toCSR();
  261:             csrMatrix->currentNode = csrMatrix->currentNode->bottomNode;
  262:             if( csrMatrix->leftMost[ currentNode->row ]==NULL )
  263:                 csrMatrix->leftMost[ currentNode->row ] =
csrMatrix->currentNode;
  264:             else
  265:             {
  266:                 temp = csrMatrix->leftMost[ currentNode->row ];
  267:                 while( temp->rightNode!=NULL && temp->rightNode->col <
currentNode->col )
  268:                     temp = temp->rightNode;
  269:                 csrMatrix->currentNode->rightNode = temp->rightNode;
  270:                 temp->rightNode = csrMatrix->currentNode;
  271:             }
  272:             currentNode=currentNode->bottomNode;
  273:         }
  274:     }
  275:
  276:     return csrMatrix;
  277: }
  278: void blockMatrix::insertValueCSR(int row, int col, double value )
  279: {
  280:     if( value==0 )
  281:         return;
  282:     createBlock( row/blockSize, col/blockSize );
  283:     currentNode->insertValueCSR( row%blockSize, col%blockSize, value );
  284: }
  285: void blockMatrix::insertValueCSC(int row, int col, double value )
  286: {
  287:     if( value==0 )
  288:         return;
  289:     createBlock( row/blockSize, col/blockSize );
  290:     currentNode->insertValueCSC( row%blockSize, col%blockSize, value );
  291: }
  292: void blockMatrix::blockInsertValueCSR(int bRow, int bCol, int row, int col,
double value )
  293: {
  294:     if( value==0 )
  295:         return;
  296:     createBlock( bRow, bCol );
  297:     currentNode->insertValueCSR( row%blockSize, col%blockSize, value );
  298: }
  299: void blockMatrix::blockInsertValueCSC(int bRow, int bCol, int row, int col,
double value )
  300: {
  301:     if( value==0 )
  302:         return;
  303:     createBlock( bRow, bCol );
  304:     currentNode->insertValueCSC( row%blockSize, col%blockSize, value );
  305: }
```

```
  306: double blockMatrix::getDiagonal( int b, int i )
  307: {
  308:     return diagonals[b]->getDiagonal( i );
  309: }
  310: void blockMatrix::operator=( const sparseMatrix& B )
  311: {
  312:     //Iterate through sparseMatrix elements
  313:     int col=0;
  314:     for( int k=0; k<B.nnz; k++ )
  315:     {
  316:         if( k==B.colptr[col+1] )
  317:             col++;
  318:
  319:         //-Create AND select currentNode
  320:         createBlock( B.adjncy[k]/blockSize, col/blockSize );
  321:
  322:         //-Add item to CURRENT block
  323:         currentNode->insertValueCSC( B.adjncy[k]%blockSize, col%blockSize,
B.nzval[k] );
  324:     }
  325: }
  326:
  327: void blockMatrix::createBlock( int row, int col )
  328: {
  329:     matrixNode *temp=NULL;
  330:
  331:     if( row==col ) //Block is a diagonal
  332:     {
  333:         currentNode = diagonals[ row ];
  334:         return;
  335:     }
  336:     else if( row > col )  //Insert Block below diagonal
  337:     {
  338:         currentNode = diagonals[ col ];
  339:         for(;;)
  340:         {
  341:             if( currentNode->bottomNode==NULL )
  342:             {
  343:                 currentNode->bottomNode = new matrixNode(blockSize,row,col);
  344:                 currentNode = currentNode->bottomNode;
  345:
  346:                 currentNode->sizeM = diagonals[ row ]->sizeM;
  347:                 currentNode->sizeN = diagonals[ col ]->sizeN;
  348:
  349:                 if( leftMost[row]==NULL )
  350:                     leftMost[row]=currentNode;
  351:                 else if( leftMost[row]->col > col )
  352:                 {
  353:                     currentNode->rightNode = leftMost[row];
  354:                     leftMost[row] = currentNode;
  355:                 }
  356:                 else
  357:                 {
  358:                     temp = leftMost[row];
  359:                     while( temp->rightNode!=NULL && temp->rightNode->col <
col )
  360:                         temp = temp->rightNode;
  361:                     currentNode->rightNode = temp->rightNode;
  362:                     temp->rightNode = currentNode;
  363:                 }
  364:                 return;
  365:             }
  366:             else if( currentNode->bottomNode->row == row )
```

```
  367:             {
  368:                 currentNode = currentNode->bottomNode;
  369:                 return;
  370:             }
  371:             else if( currentNode->bottomNode->row > row )
  372:             {
  373:                 matrixNode* temp = currentNode->bottomNode;
  374:                 currentNode->bottomNode = new matrixNode(blockSize,row,col);
  375:                 currentNode = currentNode->bottomNode;
  376:
  377:                 currentNode->sizeM = diagonals[ row ]->sizeM;
  378:                 currentNode->sizeN = diagonals[ col ]->sizeN;
  379:
  380:                 currentNode->bottomNode = temp;
  381:                 if( leftMost[row]==NULL )
  382:                     leftMost[row]=currentNode;
  383:                 else if( leftMost[row]->col > col )
  384:                 {
  385:                     currentNode->rightNode = leftMost[row];
  386:                     leftMost[row] = currentNode;
  387:                 }
  388:                 else
  389:                 {
  390:                     temp = leftMost[row];
  391:                     while( temp->rightNode!=NULL && temp->rightNode->col <
col )
  392:                         temp = temp->rightNode;
  393:                     currentNode->rightNode = temp->rightNode;
  394:                     temp->rightNode = currentNode;
  395:                 }
  396:                 return;
  397:             }
  398:             else //Proceed to bottomNode
  399:                 currentNode = currentNode->bottomNode;
  400:         }
  401:     }
  402:     else //row < col  //Insert Block to right of diagonal
  403:     {
  404:         currentNode = diagonals[ row ];
  405:         for(;;)
  406:         {
  407:             if( currentNode->rightNode==NULL )
  408:             {
  409:                 currentNode->rightNode = new matrixNode(blockSize,row,col);
  410:                 currentNode = currentNode->rightNode;
  411:
  412:                 currentNode->sizeM = diagonals[ row ]->sizeM;
  413:                 currentNode->sizeN = diagonals[ col ]->sizeN;
  414:
  415:                 if( topMost[col] == NULL )
  416:                     topMost[col] = currentNode;
  417:                 else if( topMost[col]->row > row )
  418:                 {
  419:                     currentNode->bottomNode = topMost[col];
  420:                     topMost[col] = currentNode;
  421:                 }
  422:                 else
  423:                 {
  424:                     temp = topMost[col];
  425:                     while( temp->bottomNode!=NULL && temp->bottomNode->row <
row )
  426:                         temp = temp->bottomNode;
  427:                     currentNode->bottomNode = temp->bottomNode;
```

```
428:                     temp->bottomNode = currentNode;
429:                 }
430:                 return;
431:             }
432:             else if ( currentNode->rightNode->col == col )
433:             {
434:                 currentNode = currentNode->rightNode;
435:                 return;
436:             }
437:             else if( currentNode->rightNode->col > col ) //Insert between
current and right  nodes
438:             {
439:                 matrixNode* temp = currentNode->rightNode;
440:                 currentNode->rightNode = new matrixNode(blockSize,row,col);
441:                 currentNode = currentNode->rightNode;
442:
443:                 currentNode->sizeM = diagonals[ row ]->sizeM;
444:                 currentNode->sizeN = diagonals[ col ]->sizeN;
445:
446:                 currentNode->rightNode = temp;
447:                 if( topMost[col] == NULL )
448:                     topMost[col] = currentNode;
449:                 else if( topMost[col]->row > row )
450:                 {
451:                     currentNode->bottomNode = topMost[col];
452:                     topMost[col] = currentNode;
453:                 }
454:                 else
455:                 {
456:                     temp = topMost[col];
457:                     while( temp->bottomNode!=NULL && temp->bottomNode->row <
row )
458:                         temp = temp->bottomNode;
459:                     currentNode->bottomNode = temp->bottomNode;
460:                     temp->bottomNode = currentNode;
461:                 }
462:                 return;
463:             }
464:             else //Proceed to rightNode
465:                 currentNode = currentNode->rightNode;
466:         }
467:     }
468: }
469: bool blockMatrix::hasBlock( int row, int col )
470: {
471:     matrixNode *temp;
472:     if( row==col )
473:         return true;
474:     else if( row > col )
475:     {
476:         temp = diagonals[col]->bottomNode;
477:         while( temp!=NULL && temp->row < row )
478:             temp = temp->bottomNode;
479:         if( temp!=NULL && temp->row==row )
480:             return true;
481:     }
482:     else
483:     {
484:         temp = diagonals[row]->rightNode;
485:         while( temp!=NULL && temp->col < col )
486:             temp = temp->rightNode;
487:         if( temp!=NULL && temp->col==col )
488:             return true;
```

```
489:     }
490:     return false;
491: }
492: void blockMatrix::initCSCNodes()
493: {
494:     int r,c;
495:     double v;
496:     for( int i=0; i<bm; i++ )
497:     {
498:         currentNode = diagonals[ i ];
499:         while( currentNode!= NULL )
500:         {
501:             currentNode->getFirstCSC(r,c,v);
502:             currentNode=currentNode->rightNode;
503:         }
504:         currentNode = diagonals[ i ]->bottomNode;
505:         while( currentNode!= NULL )
506:         {
507:             currentNode->getFirstCSC(r,c,v);
508:             currentNode=currentNode->bottomNode;
509:         }
510:     }
511: }
512: void blockMatrix::initCSRNodes()
513: {
514:     int r,c;
515:     double v;
516:     for( int i=0; i<bm; i++ )
517:     {
518:         currentNode = diagonals[ i ];
519:         while( currentNode!= NULL )
520:         {
521:             currentNode->getFirstCSR(r,c,v);
522:             currentNode=currentNode->rightNode;
523:         }
524:         currentNode = diagonals[ i ]->bottomNode;
525:         while( currentNode!= NULL )
526:         {
527:             currentNode->getFirstCSR(r,c,v);
528:             currentNode=currentNode->bottomNode;
529:         }
530:     }
531: }
532: void blockMatrix::dump()
533: {
534:     printf("  m: %d\n",m);
535:     printf("  n: %d\n",n);
536:     printf("  bm: %d\n",bm);
537:     printf("  bn: %d\n",bn);
538:     printf("  blockSize: %d\n",blockSize);
539:
540:     printf("  leftMost:\n");
541:     for( int i=0; i<bm; i++ )
542:         if( leftMost[i]!=NULL )
543:             printf("    L[%d][%d]\n",leftMost[i]->row,leftMost[i]->col);
544:     printf("  topMost:\n");
545:     for( int i=0; i<bm; i++ )
546:         if( topMost[i]!=NULL )
547:             printf("    T[%d][%d]\n",topMost[i]->row,topMost[i]->col);
548:
549:     for( int i=0; i<bm; i++ )
550:     {
551:         currentNode = diagonals[ i ];
```

```
552:         while( currentNode!= NULL )
553:         {
554:             currentNode->dump();
555:             currentNode=currentNode->rightNode;
556:         }
557:         currentNode = diagonals[ i ]->bottomNode;
558:         while( currentNode!= NULL )
559:         {
560:             currentNode->dump();
561:             currentNode=currentNode->bottomNode;
562:         }
563:     }
564: }
565:
566: CSFBlock* blockMatrix::toCSCBlock()
567: {
568:     CSFBlock* cscBlock = new CSFBlock(m,n);
569:     assert(cscBlock!=NULL);
570:
571:     int r,c;
572:     double v;
573:
574:     initCSCNodes();
575:
576:     for( int i=0; i<bm; i++ )
577:     {
578:         for( int j=0; j<diagonals[i]->sizeM; j++ )
579:         {
580:             if( topMost[i]!=NULL )
581:                 currentNode = topMost[ i ];
582:             else
583:                 currentNode = diagonals[i];
584:
585:             while( currentNode!=NULL )
586:             {
587:                 if( !currentNode->isEnd() )
588:                 {
589:                     currentNode->getCurrent(r,c,v);
590:
591:                     while( c==j && !currentNode->isEnd())
592:                     {
593:                         cscBlock->insertCSC( currentNode->row*blockSize + r,
currentNode->col*blockSize + c, v );
594:                         currentNode->getNextCSC(r,c,v);
595:                     }
596:                 }
597:                 currentNode = currentNode->bottomNode;
598:             }
599:         }
600:     }
601:
602:     return cscBlock;
603: }
604: CSFBlock* blockMatrix::toCSRBlock()
605: {
606:     CSFBlock* csrBlock = new CSFBlock(m,n);
607:     assert(csrBlock!=NULL);
608:
609:     int r, c;
610:     double v;
611:
612:     initCSRNodes();
613:
```

```
   614:     for( int i=0; i<bm; i++ )
   615:     {
   616:         for( int j=0; j<diagonals[i]->sizeN; j++ )
   617:         {
   618:             if( leftMost[i]!=NULL )
   619:                 currentNode = leftMost[ i ];
   620:             else
   621:                 currentNode = diagonals[i];
   622:
   623:             while( currentNode!=NULL )
   624:             {
   625:                 if( !currentNode->isEnd() )
   626:                 {
   627:                     currentNode->getCurrent(r,c,v);
   628:
   629:                     while( r==j && !currentNode->isEnd() )
   630:                     {
   631:                         csrBlock->insertCSR( currentNode->row*blockSize + r,
currentNode->col*blockSize + c, v );
   632:                         currentNode->getNextCSR(r,c,v);
   633:                     }
   634:                 }
   635:                 currentNode = currentNode->rightNode;
   636:             }
   637:         }
   638:     }
   639:
   640:     return csrBlock;
   641: }
   642:
   643: void blockMatrix::printCSC()
   644: {
   645:     for( int i=0; i<bm; i++ )
   646:     {
   647:         currentNode = diagonals[ i ];
   648:         while( currentNode!= NULL )
   649:         {
   650:             currentNode->printCSC();
   651:             currentNode=currentNode->rightNode;
   652:         }
   653:         currentNode = diagonals[ i ]->bottomNode;
   654:         while( currentNode!= NULL )
   655:         {
   656:             currentNode->printCSC();
   657:             currentNode=currentNode->bottomNode;
   658:         }
   659:     }
   660: }
   661: void blockMatrix::printCSR()
   662: {
   663:     for( int i=0; i<bm; i++ )
   664:     {
   665:         currentNode = diagonals[ i ];
   666:         while( currentNode!= NULL )
   667:         {
   668:             currentNode->printCSR();
   669:             currentNode=currentNode->rightNode;
   670:         }
   671:         currentNode = diagonals[ i ]->bottomNode;
   672:         while( currentNode!= NULL )
   673:         {
   674:             currentNode->printCSR();
   675:             currentNode=currentNode->bottomNode;
```

```
676:         }
677:     }
678: }
679: int blockMatrix::nnz()
680: {
681:     int count=0;
682:     for( int i=0; i<bm; i++ )
683:     {
684:         currentNode = diagonals[ i ];
685:         while( currentNode!= NULL )
686:         {
687:             count += currentNode->nnz();
688:             currentNode=currentNode->rightNode;
689:         }
690:         currentNode = diagonals[ i ]->bottomNode;
691:         while( currentNode!= NULL )
692:         {
693:             count += currentNode->nnz();
694:             currentNode=currentNode->bottomNode;
695:         }
696:     }
697:     return count;
698: }
699: int blockMatrix::numBlocks()
700: {
701:     int count=bm;
702:     for( int i=0; i<bm; i++ )
703:     {
704:         currentNode = diagonals[i]->rightNode;
705:         while( currentNode!=NULL )
706:         {
707:             count++;
708:             currentNode = currentNode->rightNode;
709:         }
710:         currentNode = diagonals[i]->bottomNode;
711:         while( currentNode!=NULL )
712:         {
713:             count++;
714:             currentNode = currentNode->bottomNode;
715:         }
716:     }
717:     return count;
718: }
719:
720:
721:
722:
723: void matrixNode::check()
724: {
725:     block->check();
726: }
727:
728: void blockMatrix::check()
729: {
730:     //Check Blocks:
731:     for( int i=0; i<bm; i++ )
732:     {
733:         currentNode = diagonals[ i ];
734:         while( currentNode!= NULL )
735:         {
736:             currentNode->check();
737:             currentNode=currentNode->rightNode;
738:         }
```

```
  739:         currentNode = diagonals[ i ]->bottomNode;
  740:         while( currentNode!= NULL )
  741:         {
  742:             currentNode->check();
  743:             currentNode=currentNode->bottomNode;
  744:         }
  745:     }
  746:
  747:     //Check structure:
  748:     for( int i=0; i<bm; i++ )
  749:     {
  750:         currentNode = leftMost[i];
  751:         while( currentNode!=NULL )
  752:         {
  753:             if( currentNode->rightNode!=NULL && currentNode->rightNode->col
<= currentNode->col )
  754:                 fprintf( stderr, "    Out-of-order: Left[%d]\n",i );
  755:             currentNode = currentNode->rightNode;
  756:         }
  757:     }
  758:     for( int i=0; i<bm; i++ )
  759:     {
  760:         currentNode = topMost[i];
  761:         while( currentNode!=NULL )
  762:         {
  763:             if( currentNode->bottomNode!=NULL &&
currentNode->bottomNode->row <= currentNode->row )
  764:                 fprintf( stderr, "    Out-of-order: Top[%d],  current=%d,
bottom=%d\n",i, currentNode->row, currentNode->bottomNode->row );
  765:             currentNode = currentNode->bottomNode;
  766:         }
  767:     }
  768: }
  769:
  770:
  771: bool matrixNode::operator!=( const matrixNode &B ) const
  772: {
  773:     if( blockSize!=B.blockSize )
  774:     {
  775:         printf("    BlockSizes don't match\n");
  776:         return true;
  777:     }
  778:     if( row!=B.row || col!=B.col )
  779:     {
  780:         printf("    Different block position!\n");
  781:         return true;
  782:     }
  783:     if( sizeM!=B.sizeM || sizeN!=B.sizeN )
  784:     {
  785:         printf("    Boundary difference!\n");
  786:         return true;
  787:     }
  788:     return (*block!=*B.block);
  789: }
  790:
```

```
 1: /*
 2:     Author: Esteban Torres
 3:     Date: August 2011
 4:     File: blockmatrix.h
 5:     Purpose: This file contains the definition for the block matrix
 6:              data structure.
 7: */
 8: #ifndef _BLOCK_MATRIX_H_
 9: #define _BLOCK_MATRIX_H_
10: #include "datastructures.h"
11: #include <vector>
12: using namespace std;
13:
14: class matrixNode; //Forward declaration
15: class nodeListNode
16: {
17:     public:
18:         nodeListNode():node(NULL),next(NULL){};
19:         nodeListNode(matrixNode*theNode):node(theNode),next(NULL){};
20:         matrixNode *node;
21:         nodeListNode *next;
22:
23: };
24: class nodeList
25: {
26:     public:
27:         nodeList():first(NULL),last(NULL){};
28:         ~nodeList()
29:         {
30:             nodeListNode *temp = first;
31:             while(temp!=NULL)
32:             {
33:                 first = first->next;
34:                 delete temp;
35:                 temp = first;
36:             }
37:             first = NULL;
38:             last  = NULL;
39:         }
40:         nodeListNode *first;
41:         nodeListNode *last;
42:         void insert(matrixNode*node)
43:         {
44:             if( first==NULL )
45:             {
46:                 first = new nodeListNode(node);
47:                 last  = first;
48:             }
49:             else
50:             {
51:                 last->next = new nodeListNode(node);
52:                 last = last->next;
53:             }
54:         }
55: };
56:
57:
58: class matrixNode
59: {
60:     private:
61:         unsigned int blockSize;
62:     public:
63:         int row; //position in the matrix. Add block row or col to get
```

```
actual index
   64:         int col; //position in the matrix
   65:
   66:         matrixNode* rightNode;
   67:         matrixNode* bottomNode;
   68:
   69:         CSFBlock *block;
   70:
   71:         int sizeM; //Height
   72:         int sizeN; //Width
   73:
   74:         //Constructors and Destructors
   75:         //
   76:         matrixNode(int bsz);
   77:         matrixNode( const matrixNode &B );
   78:         matrixNode(int bsz,int r,int c);
   79:         ~matrixNode()
   80:         {
   81:             if(block!=NULL)
   82:                 delete block;
   83:         };
   84:
   85:         void operator= ( const matrixNode *B ); //Deep copy
   86:         bool operator!=( const matrixNode &B ) const;
   87:         matrixNode* toCSR();
   88:
   89:         void reserve(int size);
   90:         void setIndex(int r, int c);
   91:
   92:         void insertValueCSR(int r, int c, double value);
   93:         void insertValueCSC(int r, int c, double value);
   94:
   95:
   96:         // Get elements
   97:         //
   98:         bool isEnd();
   99:         double atCSR( int row, int col );
  100:         double atCSC( int row, int col );
  101:
  102:         void getFirstCSC( int &r, int &c, double &v );
  103:         bool getNextCSC( int &r, int &c, double &v );
  104:
  105:         void getFirstCSR( int &r, int &c, double &v );
  106:         bool getNextCSR( int &r, int &c, double &v );
  107:
  108:         void getCurrent( int &r, int &c, double &v );
  109:         double getDiagonal( int i );
  110:
  111:         // Output
  112:         //
  113:         void dump() const;
  114:         void printCSC() const;
  115:         void printCSR() const;
  116:         int nnz();
  117:
  118:         void check();
  119: };
  120: class blockMatrix
  121: {
  122:     public:
  123:         int m;
  124:         int n;
  125:         int bm;  //Number of Blocks per Column
```

```
  126:         int bn;  //Number of Blocks per Row
  127:         int blockSize; //blockSize
  128:
  129:         matrixNode *currentNode;
  130:
  131:         /*vector<matrixNode*>  diagonals;
  132:         vector<matrixNode*> leftMost;
  133:         vector<matrixNode*> topMost;*/
  134:
  135:         matrixNode **diagonals;
  136:         matrixNode **leftMost;
  137:         matrixNode **topMost;
  138:
  139:         // Constructors and Destructors
  140:         //
  141:         blockMatrix(int bsz, int mm, int nn);
  142:         blockMatrix(blockMatrix &LM);
  143:         ~blockMatrix();
  144:
  145:         blockMatrix* toCSR();
  146:         void insertValueCSR( int row, int col, double value );
  147:         void insertValueCSC( int row, int col, double value );
  148:         void blockInsertValueCSR( int bRow, int bCol, int row, int col,
double value );
  149:         void blockInsertValueCSC( int bRow, int bCol, int row, int col,
double value );
  150:         double getDiagonal( int b, int i );
  151:         void operator= ( const sparseMatrix& B );  //Partition a sparseMatrix
  152:         void createBlock( int row, int col );  //Creates and selects block
  153:         bool hasBlock( int row, int col );
  154:
  155:         void initCSCNodes();
  156:         void initCSRNodes();
  157:
  158:         CSFBlock* toCSCBlock();
  159:         CSFBlock* toCSRBlock();
  160:
  161:         // Output
  162:         //
  163:         void dump();
  164:         void printCSC();
  165:         void printCSR();
  166:         int nnz();
  167:         int numBlocks();
  168:
  169:         void check();
  170: };
  171:
  172:
  173: #endif
  174:
```

```
/*
    Author:  Esteban Torres
    Date:    April 2012
    File:    bsld.h
    Purpose: Compute the LU Decomposition with partial pivoting or
              threshold pivoting.
             1. Partition the sparse matrix into sparse blocks.
             2. Compute the LU decomposition on a diagonal block
                2.1 Transform the sparse diagonal block to dense form.
                2.2 Compute LU decomposition
                2.3 Transform the dense L and U blocks to sparse form.
             3. Compute the LU decomposition of the Lower and Upper blocks.
*/

#ifndef _BSLD_H_
#define _BSLD_H_

#ifdef _MPI
#include <mpi.h>
#endif

#ifdef _OPENMP
#include <omp.h>
#endif

#include "blockmatrix.h"
#include "datastructures.h"
#include <limits>

/*
#include <common.h>
//#include <cblas.h>
extern "C"
{
   #include <cblas.h>
}
*/
#include <gsl/gsl_cblas.h>

class t_bsld_stats
{
    public:
        double epsilon; //Machine Precision
        double norm;    //Matrix Norm
        double norm1;
        double normI;
        double tinyPivot;
        bool replaced;
};

t_bsld_stats bsld_stats;

#ifdef _MPI
    int PROCESS_RANK = 0;
    int NUM_PROCS = 0;
    int NUM_PROCS_L = 0;
    int NUM_PROCS_U = 0;
#endif

unsigned int *BSLD_Pivots = NULL;
bool BSLD_Reordered = false;


```

```
 64: #ifdef _MPI
 65: void mpi_bcast_send_diagonal( const matrixNode &B, int source )
 66: {
 67:     int node_info[3];
 68:     double *values;
 69:     int *indices;
 70:     int *pointers;
 71:     int nnz;
 72:
 73:     node_info[0] = B.row;
 74:     node_info[1] = B.col;
 75:     node_info[2] = nnz = B.block->nnz();
 76:
 77:     MPI::COMM_WORLD.Bcast( &node_info, 3, MPI_INT, source );
 78:
 79:     values = new double[ nnz ];
 80:     for( int i=0; i<nnz; i++ )
 81:         values[i] = B.block->V[i];
 82:     MPI::COMM_WORLD.Bcast( values, nnz, MPI_DOUBLE, source );
 83:     delete [] values;
 84:     values = NULL;
 85:
 86:     indices = new int[ nnz ];
 87:     for( int i=0; i<nnz; i++ )
 88:         indices[i] = B.block->I[i];
 89:     MPI::COMM_WORLD.Bcast( indices, nnz, MPI_INT, source );
 90:     delete [] indices;
 91:     indices = NULL;
 92:
 93:     pointers = new int[B.block->n+1];
 94:     for( int i=0; i<B.block->n+1; i++ )
 95:         pointers[i] = B.block->P[i];
 96:     MPI::COMM_WORLD.Bcast( pointers, B.block->n+1, MPI_INT, source );
 97:     delete [] pointers;
 98:     pointers = NULL;
 99: }
100:
101: void mpi_bcast_recv_diagonal( matrixNode &B, int source )
102: {
103:     int node_info[3];
104:     double *values;
105:     int *indices;
106:     int *pointers;
107:     int nnz;
108:
109:     MPI::COMM_WORLD.Bcast( &node_info, 3, MPI_INT, source );
110:
111:     B.row = node_info[0];
112:     B.col = node_info[1];
113:     nnz   = node_info[2];
114:     B.block->reserve( nnz );
115:
116:     values = new double[ nnz ];
117:     MPI::COMM_WORLD.Bcast( values, nnz, MPI_DOUBLE, source );
118:     for( int i=0; i<nnz; i++ )
119:         B.block->V.push_back( values[i] );
120:     delete [] values;
121:     values = NULL;
122:
123:     indices = new int[ nnz ];
124:     MPI::COMM_WORLD.Bcast( indices, nnz, MPI_INT, source );
125:     for( int i=0; i<nnz; i++ )
126:         B.block->I.push_back( indices[i] );
```

```
127:     delete [] indices;
128:     indices = NULL;
129:
130:     pointers = new int[ B.block->n+1 ];
131:     MPI::COMM_WORLD.Bcast( pointers, B.block->n+1, MPI_INT, source );
132:     for( int i=0; i<B.block->n+1; i++ )
133:         B.block->P[i] = pointers[i];
134:     delete [] pointers;
135:     pointers = NULL;
136: }
137:
138: void mpi_bcast_send_pivots( int d, int blockSize, int size, int source )
139: {
140:     unsigned int* pivots;
141:     int pivotStart = d*blockSize;
142:
143:     pivots = new unsigned int[size];
144:     for( int i=0; i<size; i++ )
145:         pivots[i] = BSLD_Pivots[pivotStart+i];
146:     MPI::COMM_WORLD.Bcast( pivots, size, MPI_UNSIGNED, source );
147:     delete [] pivots;
148:     pivots = NULL;
149: }
150:
151: void mpi_bcast_recv_pivots( int d, int blockSize, int size, int source )
152: {
153:     unsigned int *pivots;
154:     int pivotStart = d*blockSize;
155:
156:     pivots = new unsigned int[ size ];
157:     MPI::COMM_WORLD.Bcast( pivots, size, MPI_UNSIGNED, source );
158:     for( int i=0; i<size; i++ )
159:         BSLD_Pivots[pivotStart+i] = pivots[i];
160:     delete [] pivots;
161:     pivots = NULL;
162: }
163:
164: void mpi_bcast_recv_adjacent( blockMatrix &LL, blockMatrix &UU, int source )
165: {
166:     int node_info[3];
167:     double *values;
168:        int *indices;
169:        int *pointers;
170:        int nnz;
171:     matrixNode *currentNode = NULL;
172:
173:     for( ;; )
174:     {
175:         MPI::COMM_WORLD.Bcast( &node_info, 3, MPI_INT, source );
176:         nnz = node_info[2];
177:
178:         if( nnz!=0 )
179:         {
180:             if( node_info[0] > node_info[1] ) //Lower
181:             {
182:                 LL.createBlock( node_info[0], node_info[1] );
183:                 currentNode = LL.currentNode;
184:             }
185:             else //Upper
186:             {
187:                 UU.createBlock( node_info[0], node_info[1] );
188:                 currentNode = UU.currentNode;
189:             }
```

```
  190:
  191:             currentNode->block->reserve(nnz);
  192:             values = new double[ nnz ];
  193:             MPI::COMM_WORLD.Bcast( values, nnz, MPI_DOUBLE, source );
  194:             for( int i=0; i<nnz; i++ )
  195:                 currentNode->block->V.push_back( values[i] );
  196:             delete [] values;
  197:             values = NULL;
  198:
  199:             indices = new int[ nnz ];
  200:             MPI::COMM_WORLD.Bcast( indices, nnz, MPI_INT, source );
  201:             for( int i=0; i<nnz; i++ )
  202:                 currentNode->block->I.push_back( indices[i] );
  203:             delete [] indices;
  204:             indices = NULL;
  205:
  206:             pointers = new int[ currentNode->block->n+1 ];
  207:             MPI::COMM_WORLD.Bcast( pointers, currentNode->block->n+1,
MPI_INT, source );
  208:             for( int i=0; i<currentNode->block->n+1; i++ )
  209:                 currentNode->block->P[i] = pointers[i];
  210:             delete [] pointers;
  211:             pointers = NULL;
  212:         }
  213:         else
  214:         {
  215:             break;
  216:         }
  217:     }
  218: }
  219: void mpi_bcast_send_lower( blockMatrix &LL, int d, int source )
  220: {
  221:     int node_info[3];
  222:     double *values;
  223:        int *indices;
  224:        int *pointers;
  225:        int nnz;
  226:
  227:     matrixNode *B = LL.diagonals[d]->bottomNode;
  228:
  229:     while( B!=NULL )
  230:     {
  231:         node_info[0] = B->row;
  232:         node_info[1] = B->col;
  233:         node_info[2] = nnz = B->block->nnz();
  234:
  235:         MPI::COMM_WORLD.Bcast( &node_info, 3, MPI_INT, source );
  236:
  237:         values = new double[ nnz ];
  238:         for( int i=0; i<nnz; i++ )
  239:             values[i] = B->block->V[i];
  240:         MPI::COMM_WORLD.Bcast( values, nnz, MPI_DOUBLE, source );
  241:         delete [] values;
  242:         values = NULL;
  243:
  244:         indices = new int[ nnz ];
  245:         for( int i=0; i<nnz; i++ )
  246:             indices[i] = B->block->I[i];
  247:         MPI::COMM_WORLD.Bcast( indices, nnz, MPI_INT, source );
  248:         delete [] indices;
  249:         indices = NULL;
  250:
  251:         pointers = new int[B->block->n+1];
```

```
252:         for( int i=0; i<B->block->n+1; i++ )
253:             pointers[i] = B->block->P[i];
254:         MPI::COMM_WORLD.Bcast( pointers, B->block->n+1, MPI_INT, source );
255:         delete [] pointers;
256:         pointers = NULL;
257:
258:         B = B->bottomNode;
259:     }
260:
261:     node_info[0] = -1;
262:     node_info[1] = -1;
263:     node_info[2] = 0;
264:     MPI::COMM_WORLD.Bcast( &node_info, 3, MPI_INT, source );
265: }
266: void mpi_bcast_send_upper(blockMatrix &UU, int d, int source )
267: {
268:     int node_info[3];
269:     double *values;
270:        int *indices;
271:        int *pointers;
272:        int nnz;
273:
274:     matrixNode *B = UU.diagonals[d]->rightNode;
275:
276:     while( B!=NULL )
277:     {
278:         node_info[0] = B->row;
279:         node_info[1] = B->col;
280:         node_info[2] = nnz = B->block->nnz();
281:
282:         MPI::COMM_WORLD.Bcast( &node_info, 3, MPI_INT, source );
283:
284:         values = new double[ nnz ];
285:         for( int i=0; i<nnz; i++ )
286:             values[i] = B->block->V[i];
287:         MPI::COMM_WORLD.Bcast( values, nnz, MPI_DOUBLE, source );
288:         delete [] values;
289:         values = NULL;
290:
291:         indices = new int[ nnz ];
292:         for( int i=0; i<nnz; i++ )
293:             indices[i] = B->block->I[i];
294:         MPI::COMM_WORLD.Bcast( indices, nnz, MPI_INT, source );
295:         delete [] indices;
296:         indices = NULL;
297:
298:         pointers = new int[B->block->n+1];
299:         for( int i=0; i<B->block->n+1; i++ )
300:             pointers[i] = B->block->P[i];
301:         MPI::COMM_WORLD.Bcast( pointers, B->block->n+1, MPI_INT, source );
302:         delete [] pointers;
303:         pointers = NULL;
304:
305:         B = B->rightNode;
306:     }
307:
308:     node_info[0] = -1;
309:     node_info[1] = -1;
310:     node_info[2] = 0;
311:     MPI::COMM_WORLD.Bcast( &node_info, 3, MPI_INT, source );
312: }
313: #endif
314:
```

```
  315: int bsld_computeDiagonal( blockMatrix &A, blockMatrix &L, blockMatrix &U, int
 d )
  316: {
  317:     denseMatrix dA   (A.blockSize);
  318:     denseMatrix dL   (A.blockSize);
  319:     denseMatrix dU   (A.blockSize);
  320:
  321:     nodeList leftNodes;
  322:     nodeList topNodes;
  323:     nodeListNode *leftNode = NULL;
  324:     nodeListNode *topNode  = NULL;
  325:     denseMatrix dLeft   (A.blockSize);
  326:     denseMatrix dTop   (A.blockSize);
  327:
  328:     matrixNode *top, *left;
  329:
  330:     dA.fromCSC( A.diagonals[d]->block );
  331:
  332:     //pivotStart: Points to the start of the pivot vector for this block.
  333:     //
  334:     int pivotStart = d*A.blockSize;
  335:
  336:     //Rows: Keeps track of which rows have been ordered.
  337:     //Rows[i] = j ;    Row i in L is computed from row j in A
  338:     //
  339:     unsigned int *Rows = new unsigned int[A.blockSize];
  340:     assert(Rows!=NULL);
  341:     for( int i=0; i<A.blockSize; i++ )
  342:         Rows[i] = i;
  343:
  344:     double *A_prime = new double[A.diagonals[d]->sizeM];
  345:     double *A_prime_row = new double[A.diagonals[d]->sizeN];
  346:     assert(A_prime!=NULL);
  347:     assert(A_prime_row!=NULL);
  348:
  349:     top = U.topMost[d];
  350:     left = L.leftMost[d];
  351:     while( top!=NULL && top->row!=d && left!=NULL && left->col!=d )
  352:     {
  353:         if( top->row==left->col )
  354:         {
  355:             leftNodes.insert(left);
  356:             topNodes.insert(top);
  357:
  358:             left = left->rightNode;
  359:             top = top->bottomNode;
  360:         }
  361:         else if( top->row > left->col )
  362:             left = left->rightNode;
  363:         else
  364:             top = top->bottomNode;
  365:     }
  366:
  367:     for( int i=0; i<A.diagonals[d]->sizeM; i++ )
  368:     {
  369:         int Row = Rows[i];
  370:
  371:         //Initialize A_prime and A_prime_row
  372:         //
  373:         for( int j=0; j<A.diagonals[d]->sizeN; j++ )
  374:             A_prime_row[j]=0;
  375:         for( int j=0; j<A.diagonals[d]->sizeM; j++ )
  376:             A_prime[j]=0;
```

```
377:
378:         //Compute A_prime for column
379:         //
380:         leftNode = leftNodes.first;
381:         topNode  = topNodes.first;
382:         while( leftNode!=NULL && topNode!=NULL )
383:         {
384:             dLeft.fromCSC( leftNode->node->block );
385:             dTop.fromCSR(  topNode->node->block  );
386:
387:             for( int j=i; j<A.diagonals[d]->sizeM; j++ )
388:             for( int k=0; k<A.blockSize; k++ )
389: #ifdef _BSLD_ZERO_CHECK
390:                 if( dLeft[Rows[j]][k]!=0 && dTop[k][i]!=0 ) ///ZERO CHECK
391: #endif
392:                     A_prime[Rows[j]] += dLeft[Rows[j]][k]*dTop[k][i];
393:
394:             dLeft.clear();
395:             dTop.clear();
396:
397:             leftNode = leftNode->next;
398:             topNode  = topNode->next;
399:         }
400:
401:         //Compute A_prime downward from diagonal
402:         //
403:         for( int k=i; k<A.diagonals[d]->sizeM; k++ )
404:         for( int j=0; j<i; j++ )
405:                 A_prime[Rows[k]] += dL[Rows[k]][j]*dU[j][i];
406:         for( int j=i; j<A.diagonals[d]->sizeM; j++ )
407:             A_prime[Rows[j]] = dA[Rows[j]][i] - A_prime[Rows[j]];
408:
409:
410:         //Pivoting
411:         //
412: #ifdef BSLD_THRESHOLD_PIVOTING
413:         if( abs(A_prime[ Row ]) < abs(bsld_stats.tinyPivot) )
414: #elif defined BSLD_PARTIAL_PIVOTING
415:         if( A_prime[ Row ] == 0 )
416: #endif
417:         {
418:             int iRow = i;
419:             for( int ii=i+1; ii<A.diagonals[d]->sizeM; ii++ )
420:             {
421:                 if( abs(A_prime[Rows[ii]]) > abs(A_prime[Row]) )
422:                 {
423:                     Row =  Rows[ii];
424:                     iRow = ii;
425:                 }
426:             }
427:
428:             if( iRow!=i )
429:             {
430:                 BSLD_Reordered = true;
431:                 Rows[iRow] = Rows[i];
432:                 Rows[i]    = Row;
433:             }
434:         }
435:
436:         //Replace tiny pivots
437:         //
438:         if( abs(A_prime[ Row ]) < abs(bsld_stats.tinyPivot) )
439:         {
```

```
440:             A_prime[ Row ] = bsld_stats.tinyPivot;
441:
442:             bsld_stats.replaced = true;
443:
444:             if( A_prime[ Row ] == 0 )
445:                 return -1;
446:         }
447:
448:         BSLD_Pivots[pivotStart+i] = Row;
449:
450:
451:         dL[Row][i] = 1;
452:         dU[i][i] = A_prime[ Row ];
453:
454:         //Lower
455:         //
456:         for( int j=i+1; j<A.diagonals[d]->sizeM; j++ )
457:             dL[Rows[j]][i] = A_prime[Rows[j]] / A_prime[Row];
458:
459:
460:         leftNode = leftNodes.first;
461:         topNode  = topNodes.first;
462:         while( leftNode!=NULL && topNode!=NULL )
463:         {
464:             dLeft.fromCSC( leftNode->node->block );
465:             dTop.fromCSR(  topNode->node->block  );
466:
467:             for( int j=i+1; j<A.diagonals[d]->sizeN; j++ )
468:             for( int k=0; k<A.blockSize; k++ )
469:             {
470: #ifdef _BSLD_ZERO_CHECK
471:                 if( dLeft[Row][k]!=0 && dTop[k][j]!=0 ) ///ZERO CHECK
472: #endif
473:                     A_prime_row[ j ] += dLeft[Row][k]*dTop[k][j];
474:             }
475:             dLeft.clear();
476:             dTop.clear();
477:
478:             leftNode = leftNode->next;
479:             topNode  = topNode->next;
480:         }
481:
482:         for( int j=i+1; j<A.diagonals[d]->sizeN; j++ )
483:         {
484:             for( int k=0; k<i; k++ )
485:             {
486: #ifdef _BSLD_ZERO_CHECK
487:                 if( dL[Row][k]!=0 && dU[k][j]!=0 ) ///ZERO CHECK
488: #endif
489:                     A_prime_row[ j ] += dL[Row][k]*dU[k][j];
490:             }
491: #ifdef _BSLD_ZERO_CHECK
492:             if( dA[Row][j]!=0 || A_prime_row[j]!=0 ) ///ZERO CHECK
493: #endif
494:                 dU[i][j] =   dA[Row][j] - A_prime_row[j];
495:         }
496:     }
497:     delete [] Rows;
498:     delete [] A_prime;
499:     delete [] A_prime_row;
500:
501:     L.diagonals[d]->block->fromDenseCSC( dL );
502:     U.diagonals[d]->block->fromDenseCSR( dU );
```

```
503:     return 0;
504: }
505:
506: void bsld_lower( blockMatrix &A, blockMatrix &L, blockMatrix &U, int d )
507: {
508:     denseMatrix LxU (A.blockSize);
509:     denseMatrix dA   (A.blockSize);
510:     denseMatrix dL   (A.blockSize);
511:     denseMatrix dU   (A.blockSize);
512:
513:     matrixNode *currentA=NULL, *currentL=NULL, *top=NULL, *left=NULL;
514:
515:     int sizeM;
516:
517:     currentA = A.diagonals[d]->bottomNode;
518:
519:     while( currentA != NULL )
520:     {
521: #ifdef _MPI
522:         if( (currentA->row % NUM_PROCS_L) + 1 == PROCESS_RANK )
523:         {
524: #endif
525:             top = U.topMost[d];
526:             left = L.leftMost[ currentA->row ];
527:
528:             sizeM = A.diagonals[ currentA->row ]->sizeM;
529:
530:             while( top!=NULL && top->row < d && left!=NULL && left->col < d )
531:             {
532:                 if( top->row == left->col )
533:                 {
534:                     dL.fromCSC( left->block );
535:                     dU.fromCSR( top->block  );
536:
537: //--------------------
538:                     //
539:                     // LxU += dL*dU
540:                     //
541:                     cblas_dgemm(
542:                         CblasRowMajor, CblasNoTrans, CblasNoTrans,
543:                         sizeM, A.blockSize, A.blockSize,
544:                         1, dL.getPointer(),   A.blockSize,
545:                            dU.getPointer(),   A.blockSize,
546:                         1, LxU.getPointer(), A.blockSize );
547: //--------------------
548:
549:                     dL.clear();
550:                     dU.clear();
551:
552:                     left = left->rightNode;
553:                     top = top->bottomNode;
554:                 }
555:                 else if( top->row > left->col )
556:                     left = left->rightNode;
557:                 else
558:                     top = top->bottomNode;
559:             }
560:
561:             dA.fromCSC( currentA->block );
562:             dU.fromCSR( U.diagonals[d]->block );
563:
564: //----------
565:             for( int i=0; i<A.blockSize; i++ )
```

```
  566:             for( int j=0; j<sizeM; j++ )
  567:             {
  568:                 for( int k=0; k<i; k++ )
  569: #ifdef _BSLD_ZERO_CHECK
  570:                     if( dL[j][k]!=0 && dU[k][i]!=0 ) ///ZERO CHECK
  571: #endif
  572:                         LxU[j][i] += dL[j][k]*dU[k][i];
  573:
  574: #ifdef _BSLD_ZERO_CHECK
  575:                 if( dA[j][i]!=0 || LxU[j][i]!=0 ) ///ZERO CHECK
  576: #endif
  577:                     dL[j][i] = ( dA[j][i] - LxU[j][i] ) / dU[i][i];
  578:             }
  579: //----------
  580:
  581:             if( dL.hasNNZ() )
  582:             {
  583:                 #pragma omp critical
  584:                 {
  585:                     L.createBlock( currentA->row, currentA->col );
  586:                     currentL = L.currentNode;
  587:                 }
  588:                 currentL->block->fromDenseCSC( dL );
  589:             }
  590:
  591:             LxU.clear();
  592:             dA.clear();
  593:             dL.clear();
  594:             dU.clear();
  595: #ifdef _MPI
  596:          }
  597: #endif
  598:         currentA=currentA->bottomNode;
  599:     }
  600: }
  601:
  602: void bsld_fillLower( blockMatrix &A, blockMatrix &L, blockMatrix &U, int d )
  603: {
  604:     denseMatrix LxU (A.blockSize);
  605:     denseMatrix dL   (A.blockSize);
  606:     denseMatrix dU   (A.blockSize);
  607:
  608:     matrixNode *top=NULL, *left=NULL, *currentL=NULL;
  609:
  610:     for( int i=d+1; i<L.bm; i++ )
  611:     {
  612: #ifdef _MPI
  613:         if( (i%NUM_PROCS_L)+1 != PROCESS_RANK )
  614:             continue;
  615: #endif
  616:         top  = U.topMost[d];
  617:         left = L.leftMost[i];
  618:
  619:         if( left!=NULL && top!=NULL && !A.hasBlock(left->row, d) )
  620:         {
  621:             while( left!=NULL && left->col < d  && top!=NULL && top->row < d
)
  622:             {
  623:                 if( left->col==top->row )
  624:                 {
  625:                     dL.fromCSC( left->block );
  626:                     dU.fromCSR( top->block  );
  627:
```

```
628: //--------------------
629:                     //
630:                     // LxU += dL*dU
631:                     //
632:                     cblas_dgemm(
633:                         CblasRowMajor, CblasNoTrans, CblasNoTrans,
634:                         left->sizeM, A.blockSize, A.blockSize,
635:                         1, dL.getPointer(),   A.blockSize,
636:                            dU.getPointer(),   A.blockSize,
637:                         1, LxU.getPointer(), A.blockSize );
638: //--------------------
639:
640:                     dL.clear();
641:                     dU.clear();
642:
643:                     left = left->rightNode;
644:                     top = top->bottomNode;
645:                 }
646:                 else if( left->col > top->row )
647:                     top = top->bottomNode;
648:                 else
649:                     left = left->rightNode;
650:             }
651:
652:             if( LxU.hasNNZ() )
653:             {
654:                 dU.fromCSR( U.diagonals[d]->block );
655:
656: //--------------
657:                 for( int ii=0; ii<A.blockSize; ii++ )
658:                 for( int j=0; j<A.diagonals[ d ]->sizeM; j++ )
659:                 {
660:                     for( int k=0; k<ii; k++ )
661: #ifdef _BSLD_ZERO_CHECK
662:                         if( dL[j][k]!=0 && dU[k][ii]!=0 ) ///ZERO CHECK
663: #endif
664:                             LxU[j][ii] += dL[j][k]*dU[k][ii];
665:
666: #ifdef _BSLD_ZERO_CHECK
667:                     if( LxU[j][ii]!=0 ) ///ZERO CHECK
668: #endif
669:                         dL[j][ii] = ( 0 - LxU[j][ii] ) / dU[ii][ii];
670:                 }
671: //--------------
672:
673:                 if( dL.hasNNZ() )
674:                 {
675:                     #pragma omp critical
676:                     {
677:                         L.createBlock( i, d );
678:                         currentL = L.currentNode;
679:                     }
680:                     currentL->block->fromDenseCSC( dL );
681:                 }
682:                 dL.clear();
683:                 dU.clear();
684:             }
685:             LxU.clear();
686:         }
687:     }
688: }
689: void bsld_upper( blockMatrix &A, blockMatrix &L, blockMatrix &U, int d )
690: {
```

```
691:     denseMatrix LxU (A.blockSize);
692:     denseMatrix dA   (A.blockSize);
693:     denseMatrix dL   (A.blockSize);
694:     denseMatrix drL   (A.blockSize);
695:     denseMatrix dU   (A.blockSize);
696:     matrixNode *currentA=NULL, *currentU=NULL, *top=NULL, *left=NULL;
697:
698:     currentA = A.diagonals[d]->rightNode;
699:
700:     //pivotStart: Points to the start of the pivot vector for this block.
701:     //
702:     int pivotStart = d*A.blockSize;
703:
704:     int sizeN;
705:
706:     while( currentA != NULL )
707:     {
708: #ifdef _MPI
709:         if( (currentA->col % NUM_PROCS_U) + NUM_PROCS_L + 1 == PROCESS_RANK )
710:         {
711: #endif
712:             top = U.topMost[ currentA->col ];
713:             left = L.leftMost[d];
714:
715:             sizeN = A.diagonals[ currentA->col ]->sizeN;
716:
717:             while( top!=NULL && top->row < d && left!=NULL && left->col < d )
718:             {
719:                 if( top->row == left->col )
720:                 {
721:                     dL.fromCSC( left->block );
722:                     dU.fromCSR( top->block  );
723:
724: //--------------------
725:                     // Reorder [L] first.
726:                     //
727:                     for( int i=0; i<A.blockSize; i++ )
728:                     for( int j=0; j<A.blockSize; j++ )
729:                         drL[i][j] = dL[ BSLD_Pivots[pivotStart+i]][j];
730:
731:                     //
732:                     // LxU += dL*dU
733:                     //
734:                     cblas_dgemm(
735:                         CblasRowMajor, CblasNoTrans, CblasNoTrans,
736:                         A.blockSize, sizeN, A.blockSize,
737:                         1, drL.getPointer(),   A.blockSize,
738:                            dU.getPointer(),   A.blockSize,
739:                         1, LxU.getPointer(), A.blockSize );
740: //--------------------
741:
742:                     drL.clear();
743:                     dL.clear();
744:                     dU.clear();
745:
746:                     left = left->rightNode;
747:                     top = top->bottomNode;
748:                 }
749:                 else if( top->row > left->col )
750:                     left = left->rightNode;
751:                 else
752:                     top = top->bottomNode;
753:             }
```

```
  754:
  755:             dA.fromCSC( currentA->block );
  756:             dL.fromCSC( L.diagonals[d]->block );
  757:
  758:             for( int i=0; i<A.blockSize; i++ )
  759:             for( int j=0; j<sizeN; j++ )
  760:             {
  761:                 for( int k=0; k<i; k++ )
  762: #ifdef _BSLD_ZERO_CHECK
  763:                     if( dL[BSLD_Pivots[pivotStart+i]][k]!=0 && dU[k][j]!=0 )
///ZERO CHECK
  764: #endif
  765:                         LxU[i][j] +=
dL[BSLD_Pivots[pivotStart+i]][k]*dU[k][j];
  766:
  767: #ifdef _BSLD_ZERO_CHECK
  768:                 if( dA[BSLD_Pivots[pivotStart+i]][j]!=0 || LxU[i][j]!=0 )
///ZERO CHECK
  769: #endif
  770:                     dU[i][j] =   dA[BSLD_Pivots[pivotStart+i]][j] -
LxU[i][j];
  771:             }
  772:
  773:             if( dU.hasNNZ() )
  774:             {
  775:                 #pragma omp critical
  776:                 {
  777:                     U.createBlock( currentA->row, currentA->col );
  778:                     currentU = U.currentNode;
  779:                 }
  780:                 currentU->block->fromDenseCSR( dU );
  781:             }
  782:
  783:             LxU.clear();
  784:             dA.clear();
  785:             dL.clear();
  786:             dU.clear();
  787: #ifdef _MPI
  788:          }
  789: #endif
  790:         currentA = currentA->rightNode;
  791:     }
  792: }
  793: void bsld_fillUpper( blockMatrix &A, blockMatrix &L, blockMatrix &U, int d )
  794: {
  795:     denseMatrix LxU (A.blockSize);
  796:     denseMatrix dL   (A.blockSize);
  797:     denseMatrix drL   (A.blockSize);
  798:     denseMatrix dU   (A.blockSize);
  799:
  800:     matrixNode *top=NULL, *left=NULL, *currentU=NULL;
  801:
  802:     //pivotStart: Points to the start of the pivot vector for this block.
  803:     //
  804:     int pivotStart = d*A.blockSize;
  805:
  806:     for( int u=d+1; u<U.bm; u++ )
  807:     {
  808: #ifdef _MPI
  809:         if( (u%NUM_PROCS_U)+NUM_PROCS_L+1 != PROCESS_RANK )
  810:             continue;
  811: #endif
  812:         top = U.topMost[u];
```

```
  813:         left = L.leftMost[d];
  814:
  815:         if( top!=NULL && left!=NULL && !A.hasBlock(d, top->col) )
  816:         {
  817:             while( left!=NULL && left->col < d  && top!=NULL && top->row < d
)
  818:             {
  819:                 if( left->col==top->row )
  820:                 {
  821:                     dL.fromCSC( left->block );
  822:                     dU.fromCSR( top->block  );
  823:
  824: //--------------------
  825:                     // Reorder [L] first.
  826:                     //
  827:                     for( int i=0; i<A.blockSize; i++ )
  828:                     for( int j=0; j<A.blockSize; j++ )
  829:                         drL[i][j] = dL[ BSLD_Pivots[pivotStart+i]][j];
  830:
  831:                     //
  832:                     // LxU += dL*dU
  833:                     //
  834:                     cblas_dgemm(
  835:                         CblasRowMajor, CblasNoTrans, CblasNoTrans,
  836:                         A.blockSize, top->sizeN, A.blockSize,
  837:                         1, drL.getPointer(),   A.blockSize,
  838:                            dU.getPointer(),   A.blockSize,
  839:                         1, LxU.getPointer(), A.blockSize );
  840: //--------------------
  841:
  842:                     drL.clear();
  843:                     dL.clear();
  844:                     dU.clear();
  845:
  846:                     left = left->rightNode;
  847:                     top = top->bottomNode;
  848:                 }
  849:                 else if( left->col > top->row )
  850:                     top = top->bottomNode;
  851:                 else
  852:                     left = left->rightNode;
  853:             }
  854:
  855:             if( LxU.hasNNZ() )
  856:             {
  857:                 dL.fromCSC( L.diagonals[d]->block );
  858:
  859:                 for( int i=0; i<A.blockSize; i++ )
  860:                 for( int j=0; j<A.diagonals[ u ]->sizeN; j++ )
  861:                 {
  862:                     for( int k=0; k<i; k++ )
  863: #ifdef _BSLD_ZERO_CHECK
  864:                         if( dL[BSLD_Pivots[pivotStart+i]][k]!=0 &&
dU[k][j]!=0 ) ///ZERO CHECK
  865: #endif
  866:                             LxU[i][j] +=
dL[BSLD_Pivots[pivotStart+i]][k]*dU[k][j];
  867:
  868: #ifdef _BSLD_ZERO_CHECK
  869:                         if( LxU[i][j]!=0 ) ///ZERO CHECK
  870: #endif
  871:                             dU[i][j] = 0 - LxU[i][j];
  872:                 }
```

```
873:
874:                 if( dU.hasNNZ() )
875:                 {
876:                     #pragma omp critical
877:                     {
878:                         U.createBlock( d, u );
879:                         currentU = U.currentNode;
880:                     }
881:                     currentU->block->fromDenseCSR( dU );
882:                 }
883:
884:                 dL.clear();
885:                 dU.clear();
886:             }
887:             LxU.clear();
888:         }
889:     }
890: }
891:
892: int bsld( const sparseMatrix &A, sparseMatrix &L, sparseMatrix &U, rowPerm
&P, int blockSize )
893: {
894:     if( blockSize <=1 )
895:         return -2;
896:
897:     blockMatrix AA(blockSize,A.m,A.n),
898:                 LL(blockSize,A.m,A.n),
899:                 UU(blockSize,A.m,A.n);
900:     int dStatus = 0;
901:
902:     //Partition matrix A
903:     //
904:     AA = A;
905:
906:
907:     //Initialize pivot vector
908:     //
909:     BSLD_Pivots = new unsigned int[A.m];
910:     assert(BSLD_Pivots!=NULL);
911:     for( int i=0; i<A.m; i++ )
912:         BSLD_Pivots[i] = 0;
913:
914:
915:
916: #ifdef _MPI
917:         if( PROCESS_RANK == 0 )
918:         {
919: #endif
920:             //Compute Machine Epsilon, Matrix Norm, and Tiny Pivot
921:             //
922:             //
923:             bsld_stats.epsilon = std::numeric_limits<double>::epsilon();
924:             bsld_stats.replaced = false;
925:
926:             bsld_stats.norm  = 0;
927:             bsld_stats.norm1 = 0; //Sum of Cols
928:             bsld_stats.normI = 0; //Sum of Rows
929:
930:             {//normI [Sum of Rows]
931:                 CSFBlock cscA(A.m,A.n);
932:                 CSFBlock csrA(A.m,A.n);
933:                 cscA = A;
934:                 csrA = cscA.toCSR();
```

```
935:
936:                 for( int i=0; i<A.m; i++ )
937:                 {
938:                     double sum = 0;
939:                     for( int j=csrA.P[i]; j<csrA.P[i+1]; j++ )
940:                         sum += abs(csrA.V[j]);
941:                     bsld_stats.normI = max( bsld_stats.normI, sum );
942:                 }
943:             }
944:
945:             for( int i=0; i<A.n; i++ )
946:             {
947:                 double sum = 0;
948:                 for( int j=A.colptr[i]; j<A.colptr[i+1]; j++ )
949:                     sum += abs(A.nzval[j]);
950:                 bsld_stats.norm1 = max( bsld_stats.norm1, sum );
951:             }
952:
953:             bsld_stats.norm = bsld_stats.normI;
954:             bsld_stats.tinyPivot = sqrt(bsld_stats.epsilon) *
bsld_stats.normI ;
955:
956: #ifdef _MPI
957:         }
958: #endif
959:
960:
961:
962:
963:     for( int i=0; i<AA.bm; i++ )
964:     {
965: #ifdef _MPI
966:         if( PROCESS_RANK == 0 )
967:         {
968: #endif
969:             dStatus = bsld_computeDiagonal(AA, LL, UU, i);
970: #ifdef _MPI
971:             MPI::COMM_WORLD.Bcast( &dStatus, 1, MPI_INT, 0 );
972: #endif
973:             if( dStatus!=0 )
974:                 return dStatus;
975:
976: #ifdef _MPI
977:             //Send diagonal L & U blocks to ALL NODES
978:             //
979:             mpi_bcast_send_diagonal( *LL.diagonals[i], 0 );
980:             mpi_bcast_send_diagonal( *UU.diagonals[i], 0 );
981:             mpi_bcast_send_pivots( i, blockSize, AA.diagonals[i]->sizeM, 0 );
982:
983:             //Receive adjacent nodes
984:             //
985:             MPI::COMM_WORLD.Barrier();
986:             for( int j=1; j<NUM_PROCS; j++ )
987:             {
988:                 mpi_bcast_recv_adjacent(LL,UU,j);
989:             }
990:         }
991:         else
992:         {
993:             MPI::COMM_WORLD.Bcast( &dStatus, 1, MPI_INT, 0 );
994:             if( dStatus!=0 )
995:                 return dStatus;
996:
```

```
  997:             //Receive diagonal L & U blocks from node 0
  998:             //
  999:             mpi_bcast_recv_diagonal( *LL.diagonals[i], 0 );
 1000:             mpi_bcast_recv_diagonal( *UU.diagonals[i], 0 );
 1001:             mpi_bcast_recv_pivots( i, blockSize, AA.diagonals[i]->sizeM, 0 );
 1002:
 1003:             if( PROCESS_RANK <= NUM_PROCS/2  )
 1004:             {
 1005:                 #pragma omp parallel sections default(none)
shared(bsld_stats,AA,LL,UU,i) num_threads(2)
 1006:                 {
 1007:                     #pragma omp section
 1008:                     {
 1009:                         bsld_lower(AA,LL,UU,i);
 1010:                     }
 1011:                     #pragma omp section
 1012:                     {
 1013:                         bsld_fillLower(AA,LL,UU,i);
 1014:                     }
 1015:                 }
 1016:             }
 1017:             else
 1018:             {
 1019:                 #pragma omp parallel sections default(none)
shared(bsld_stats,AA,LL,UU,i) num_threads(2)
 1020:                 {
 1021:                     #pragma omp section
 1022:                     {
 1023:                         bsld_upper(AA,LL,UU,i);
 1024:                     }
 1025:                     #pragma omp section
 1026:                     {
 1027:                         bsld_fillUpper(AA,LL,UU,i);
 1028:                     }
 1029:                 }
 1030:             }
 1031: #else
 1032:             #pragma omp parallel sections default(none)
shared(bsld_stats,AA,LL,UU,i) num_threads(4)
 1033:             {
 1034:                 #pragma omp section
 1035:                 {
 1036:                     bsld_lower(AA,LL,UU,i);
 1037:                 }
 1038:                 #pragma omp section
 1039:                 {
 1040:                     bsld_fillLower(AA,LL,UU,i);
 1041:                 }
 1042:                 #pragma omp section
 1043:                 {
 1044:                     bsld_upper(AA,LL,UU,i);
 1045:                 }
 1046:                 #pragma omp section
 1047:                 {
 1048:                     bsld_fillUpper(AA,LL,UU,i);
 1049:                 }
 1050:             }
 1051: #endif
 1052: #ifdef _MPI
 1053:
 1054:             //Send ajacent nodes
 1055:             //
 1056:             MPI::COMM_WORLD.Barrier();
```

```
1057:             for( int j=1; j<NUM_PROCS; j++ )
1058:             {
1059:                 if( j!=PROCESS_RANK )
1060:                     mpi_bcast_recv_adjacent(LL,UU,j);
1061:                 else if( j<= NUM_PROCS/2 )
1062:                     mpi_bcast_send_lower(LL,i,j);
1063:                 else
1064:                     mpi_bcast_send_upper(UU,i,j);
1065:             }
1066:         }
1067:
1068:         MPI::COMM_WORLD.Barrier();
1069: #endif
1070:     }
1071: #ifdef _MPI
1072:     if( PROCESS_RANK == 0 )
1073:     {
1074: #endif
1075:         CSFBlock *cscL,
1076:                  *csrU,
1077:                   cscU(A.m,A.n);
1078:
1079:         cscL = LL.toCSCBlock();
1080:         csrU = UU.toCSRBlock();
1081:         cscU = csrU->toCSR();
1082:
1083:         L = *cscL;
1084:         U = cscU;
1085:
1086:         delete cscL;
1087:         delete csrU;
1088:
1089:         //Reorganize the pivoting information to:
1090:         // from: P[new] = old
1091:         //   to: P[old] = new
1092:         for( int i=0; i<A.m; i++ )
1093:         {
1094:             BSLD_Pivots[i]+=(i/blockSize)*blockSize;
1095:             P[BSLD_Pivots[i]] = i;
1096:         }
1097:         delete [] BSLD_Pivots;
1098:         BSLD_Pivots = NULL;
1099:
1100:         if( BSLD_Reordered )
1101:             L = P*L;
1102: #ifdef _MPI
1103:     }
1104: #endif
1105:     return 0;
1106: }
1107:
1108:
1109: #endif
```

```
 1: /*
 2:     Author:  Esteban Torres
 3:     Date:    July 2010
 4:     File:    datastructures.cpp
 5:     Purpose: This file contains data structure definitions and implementation
 6:              details.
 7: */
 8:
 9: #include "datastructures.h"
10: #include "blockmatrix.h"
11: #include <string.h>
12: using namespace std;
13:
14:
15: //CSFBlock-----------------------------------------------------------------
16: CSFBlock::CSFBlock(int mm, int nn):
17:     m(mm),
18:     n(nn),
19:     P(n+1),
20:     currentIndex(0),
21:     currentRow(0),
22:     currentCol(0)
23: {
24:     for( int i=0; i<n; i++ )
25:         P[i] = -1;
26:     P[n]=0;
27: }
28: CSFBlock::~CSFBlock()
29: {
30: }
31: void CSFBlock::operator= ( const CSFBlock &B )
32: {
33:     if( V.size()>0 )
34:         clear();
35:     m = B.m;
36:     n = B.n;
37:     V.reserve( B.V.size() );
38:     I.reserve( B.I.size() );
39:     P.resize( n+1 );
40:     for( unsigned int i=0; i< B.I.size(); i++ )
41:     {
42:         I.push_back( B.I[i] );
43:         V.push_back( B.V[i] );
44:     }
45:     for( int i=0; i<=B.n; i++ )
46:         P[i] = B.P[i];
47: }
48: void CSFBlock::operator= ( const sparseMatrix &B )
49: {
50:     if( V.size()>0 )
51:         clear();
52:     m = B.m;
53:     n = B.n;
54:     V.reserve( B.nnz );
55:     I.reserve( B.nnz );
56:     P.resize( n+1 );
57:     for( int i=0; i< B.nnz; i++ )
58:     {
59:         I.push_back( B.adjncy[i] );
60:         V.push_back( B.nzval[i]  );
61:     }
62:     for( int i=0; i<=B.n; i++ )
63:         P[i] = B.colptr[i];
```

```
   64: }
   65: CSFBlock CSFBlock::toCSR()
   66: {
   67:     CSFBlock csrBlock(m,n);
   68:
   69:     csrBlock.I.resize( I.size(), 0 );
   70:     csrBlock.V.resize( V.size(), 0 );
   71:
   72:     int counter=0;
   73:     vector<int> Counts(m,0);
   74:     vector<int> Counts2(m,0);
   75:     for( unsigned int i=0; i<I.size(); i++ )
   76:     {
   77:         Counts[ I[i] ] ++;
   78:         Counts2[ I[i] ] ++;
   79:     }
   80:
   81:     for( int i=0; i<m; i++ )
   82:     {
   83:         if( Counts[i]>0 )
   84:             csrBlock.P[i] = counter;
   85:         counter+=Counts[i];
   86:     }
   87:
   88:     int col=0;
   89:     int j=1;
   90:     while( P[j] < 0 )
   91:         j++;
   92:
   93:     for( int i=0; i<static_cast<int>(I.size()); i++ )
   94:     {
   95:         if( i==P[j] )
   96:         {
   97:             col=j; // not col++
   98:             while(P[++j]<0);
   99:         }
  100:         csrBlock.I[ csrBlock.P[I[i]]+(Counts[I[i]]-Counts2[I[i]]) ] = col;
  101:         csrBlock.V[ csrBlock.P[I[i]]+(Counts[I[i]]-(Counts2[I[i]]--)) ] =
V[i];
  102:     }
  103:     csrBlock.P[n] = P[n];
  104:     return csrBlock;
  105: }
  106: void CSFBlock::reserve( int size )
  107: {
  108:     V.reserve( size );
  109:     I.reserve( size );
  110: }
  111: bool CSFBlock::isEnd()
  112: {
  113:     if( currentIndex==P[n] )
  114:         return true;
  115:     return false;
  116: }
  117: double CSFBlock::atCSR(int row, int col)
  118: {
  119:     if( P[row]<0 )
  120:         return 0;
  121:     int i=P[row];
  122:     int j=row;
  123:     while( P[++j]<0 );
  124:     for( ; i<P[j]; i++ )
  125:         if( I[i]==col )
```

```
126:             return V[i];
127:     return 0;
128: }
129: double CSFBlock::atCSC(int row, int col)
130: {
131:     if( P[col]<0 )
132:         return 0;
133:     int i=P[col];
134:     int j=col;
135:     while( P[++j]<0 );
136:
137:     for( ; i<P[j]; i++ )
138:         if( I[i]==row )
139:             return V[i];
140:     return 0;
141: }
142: void CSFBlock::insertCSR(int row, int col, double value)
143: {
144:     assert(row<m);
145:     assert(col<n);
146:     if( value==0 )
147:         return;
148:
149:     if( P[row]== -1 )
150:         P[row] = V.size();
151:     V.push_back(value);
152:     I.push_back(col);
153:     P[n]++;
154: }
155: void CSFBlock::insertCSC(int row, int col, double value)
156: {
157:     assert(row<m);
158:     assert(col<n);
159:     if( value==0 )
160:         return;
161:
162:     if( P[col]== -1 )
163:         P[col] = V.size();
164:     V.push_back(value);
165:     I.push_back(row);
166:     P[n]++;
167: }
168: void CSFBlock::getFirstCSC( int &r, int &c, double &v )
169: {
170:     if( P[n]==0 )
171:     {
172:         r = -1;
173:         c = -1;
174:         v =  0;
175:         return;
176:     }
177:     c = -1;
178:     r = I[0];
179:     v = V[0];
180:     for( int j=0; j<n; j++ )
181:         if( P[j]>=0 )
182:         {
183:             c = j;
184:             break;
185:         }
186:     currentRow = r;
187:     currentCol = c;
188:     currentIndex = 0;
```

```
189: }
190: void CSFBlock::getFirstCSR( int &r, int &c, double &v )
191: {
192:     if( P[n]==0 )
193:     {
194:         r = -1;
195:         c = -1;
196:         v =  0;
197:         return;
198:     }
199:     r = -1;
200:     c = I[0];
201:     v = V[0];
202:     for( int j=0; j<n; j++ )
203:         if( P[j]>=0 )
204:         {
205:             r = j;
206:             break;
207:         }
208:     currentRow = r;
209:     currentCol = c;
210:     currentIndex = 0;
211: }
212: bool CSFBlock::getNextCSC( int &r, int &c, double &v )
213: {
214:     currentIndex++;
215:     if( currentIndex>=P[n] )
216:     {
217:         currentIndex=P[n];
218:         r=-1;
219:         c=-1;
220:         v=0;
221:         return false;
222:     }
223:
224:     currentRow = r = I[currentIndex];
225:     v = V[currentIndex];
226:
227:     int j=currentCol+1;
228:     while( P[j] < 0 )
229:         j++;
230:     if( currentIndex==P[j] )
231:         currentCol=j; //Not currentCol++
232:     c = currentCol;
233:     return true;
234: }
235: bool CSFBlock::getNextCSR( int &r, int &c, double &v )
236: {
237:     currentIndex++;
238:     if( currentIndex>=P[n] )
239:     {
240:         currentIndex=P[n];
241:         r=-1;
242:         c=-1;
243:         v=0;
244:         return false;
245:     }
246:
247:     currentCol = c = I[currentIndex];
248:     v = V[currentIndex];
249:
250:     int j=currentRow+1;
251:     while( P[j] < 0 )
```

```
252:         j++;
253:     if( currentIndex==P[j] )
254:         currentRow=j; //Not currentRow++
255:     r = currentRow;
256:     return true;
257: }
258: void CSFBlock::getCurrent( int &r, int &c, double &v )
259: {
260:     assert(currentIndex!=P[n]);
261:     r = currentRow;
262:     c = currentCol;
263:     v = V[currentIndex];
264: }
265: double CSFBlock::getDiagonal( int i )
266: {
267:     assert(i<m);
268:     int j=i+1;
269:     while( P[j] < 0 )
270:         j++;
271:     for( int x=P[i]; x<=P[j]; x++ )
272:         if( I[x]==i )
273:             return V[x];
274:
275:     return 0;
276: }
277:
278: void CSFBlock::fromDenseCSC(const denseMatrix &csc)
279: {
280:     int count=0,k=0;
281:
282:     count = csc.nnz();
283:
284:     I.resize(count,0);
285:     V.resize(count,0);
286:     P[m]=count;
287:     count = 0;
288:     for( unsigned int i=0; i<csc.n; i++ )
289:     for( unsigned int j=0; j<csc.m; j++ )
290:         if( csc[j][i]!=0 )
291:         {
292:             if( P[i]==-1 )
293:                 P[i]=count;
294:             I[k]   = j;
295:             V[k++] = csc[j][i];
296:             count++;
297:         }
298: }
299: void CSFBlock::fromDenseCSR(const denseMatrix &csr)
300: {
301:     int count=0,k=0;
302:
303:     count = csr.nnz();
304:
305:     I.resize(count,0);
306:     V.resize(count,0);
307:     P[m] = count;
308:     count = 0;
309:     for( unsigned int i=0; i<csr.m; i++ )
310:     for( unsigned int j=0; j<csr.n; j++ )
311:         if( csr[i][j]!=0 )
312:         {
313:             if( P[i]==-1 )
314:                 P[i]=count;
```

```
315:             I[k]   = j;
316:             V[k++] = csr[i][j];
317:             count++;
318:         }
319: }
320:
321: void CSFBlock::clear()
322: {
323:     currentIndex = 0;
324:     currentRow = 0;
325:     currentCol = 0;
326:     m=0;
327:     n=0;
328:     V.clear();
329:     I.clear();
330:     P.clear();
331: }
332: void CSFBlock::dump() const
333: {
334:     printf("    m: %d\n",m);
335:     printf("    n: %d\n",n);
336:     printf("      V: ");
337:     for( unsigned int i=0; i<V.size(); i++ )
338:         printf("%lf ",V[i]);
339:     printf("\n");
340:     printf("      I: ");
341:     for( unsigned int i=0; i<I.size(); i++ )
342:         printf("%3d ",I[i]);
343:     printf("\n");
344:     printf("      P: ");
345:     for( int i=0; i<n+1; i++ )
346:         printf("%3d ",P[i]);
347:     printf("\n");
348: }
349: void CSFBlock::printCSC() const
350: {
351:     double **matrix = new double* [n];
352:     assert(matrix!=NULL);
353:     for( int i=0; i<n; i++ )
354:         matrix[i] = new double [m];
355:     for( int i=0; i<m; i++ )
356:     for( int j=0; j<n; j++ )
357:         matrix[i][j] = 0;
358:
359:     int col=0;
360:     while( P[col]<0 )
361:         col++;
362:
363:     int next=col;
364:     while( P[++next]<0 );
365:
366:     for( int i=0; i<static_cast<int>(I.size()); i++ )
367:     {
368:         if( i==P[next] )
369:         {
370:             col = next;
371:             while( P[++next]<0 );
372:         }
373:         matrix[I[i]][col] = V[i];
374:     }
375:
376:     for( int i=0; i<m; i++ )
377:     {
```

```
378:         for( int j=0; j<n; j++ )
379:             if( matrix[i][j]!=0 )
380:                 printf("%4.1lf ",matrix[i][j]);
381:             else
382:                 printf("    . ");
383:         printf("\n");
384:     }
385:
386:     for( int i=0; i<n; i++ )
387:         delete[] matrix[i];
388:     delete[] matrix;
389: }
390: void CSFBlock::printCSR() const
391: {
392:     double **matrix = new double* [n];
393:     assert(matrix!=NULL);
394:     for( int i=0; i<n; i++ )
395:         matrix[i] = new double [m];
396:     for( int i=0; i<m; i++ )
397:     for( int j=0; j<n; j++ )
398:         matrix[i][j] = 0;
399:
400:     int row=0;
401:     while( P[row]<0 )
402:         row++;
403:
404:     int next=row;
405:     while( P[++next]<0 );
406:
407:     for( int i=0; i<static_cast<int>(I.size()); i++ )
408:     {
409:         if( i==P[next] )
410:         {
411:             row = next;
412:             while( P[++next]<0 );
413:         }
414:         matrix[row][I[i]] = V[i];
415:     }
416:
417:     for( int i=0; i<m; i++ )
418:     {
419:         for( int j=0; j<n; j++ )
420:             if( matrix[i][j]!=0 )
421:                 printf("%4.1lf ",matrix[i][j]);
422:             else
423:                 printf("    . ");
424:         printf("\n");
425:     }
426:
427:     for( int i=0; i<n; i++ )
428:         delete[] matrix[i];
429:     delete[] matrix;
430: }
431: int CSFBlock::nnz() const
432: {
433:     return P[n];
434: }
435:
436: void CSFBlock::check()
437: {
438:     if( I.size()!= (unsigned int)P[n] )
439:         fprintf( stderr, "    Index size error!\n" );
440:     if( V.size()!= (unsigned int)P[n] )
```

```
  441:             fprintf( stderr, "    Value size error!\n" );
  442:         for( unsigned int i=0; i<I.size(); i++ )
  443:             if( I[i]>=n )
  444:                 fprintf(stderr, "    Index out of bounds!\n");
  445:         for( int i=0; i<n; i++ )
  446:             if( P[i]==P[i+1] && P[i]>=0 )
  447:                 fprintf(stderr,"Duplicate colptrs\n");
  448:             else if( P[i]>P[i+1] && P[i]>=0 && P[i+1]>=0 )
  449:                 fprintf(stderr,"Unsorted colptr[%d]=%d  colptr[%d]=%d\n"
,i,P[i],i+1,P[i+1]);
  450:
  451:         int col=0,j=0;
  452:         while( P[col]<0 ) col++;
  453:         while( P[++j]<0 );
  454:         while( j<n )
  455:         {
  456:             for( int i=P[col]; i<P[j]-1; i++ )
  457:             {
  458:                 if( I[i]>=I[i+1] )
  459:                 {
  460:                     printf("    Elements are out of order in column %d.\n",col);
  461:                     if( m<20 )
  462:                     dump();
  463:                     return;
  464:                 }
  465:             }
  466:             while(P[++col]<0);
  467:             j=col;
  468:             while(P[++j]<0);
  469:         }
  470: }
  471:
  472: bool CSFBlock::operator!=( const CSFBlock &B ) const
  473: {
  474:         if( m!=B.m || n!=B.n )
  475:         {
  476:             printf("    Structurally different!\n");
  477:             return true;
  478:         }
  479:         if( P[n]!=B.P[B.n] )
  480:         {
  481:             printf("    Number of Elements not equal!\n");
  482:             return true;
  483:         }
  484:         for( int i=0; i<n; i++ )
  485:             if( P[i]!=B.P[i] )
  486:             {
  487:                 printf("    Pointer difference!\n");
  488:                 return true;
  489:             }
  490:         for( int i=0; i<P[n]; i++ )
  491:             if( I[i]!=B.I[i] )
  492:             {
  493:                 printf("    Index difference!\n");
  494:                 return true;
  495:             }
  496:         for( int i=0; i<P[n]; i++ )
  497:             if( abs(V[i]-B.V[i])>.000001 )
  498:             {
  499:                 printf("    Value difference: %e != %e\n",V[i],B.V[i]);
  500:                 return true;
  501:             }
  502:
```

```
  503:     return false;
  504: }
  505:
  506: //class
rowPerm--------------------------------------------------------------------
  507: rowPerm::rowPerm(int rows):nr(rows)
  508: {
  509:     rp = new int[rows];
  510: }
  511: rowPerm::~rowPerm()
  512: {
  513:     if( rp!=NULL ) delete [] rp;
  514: }
  515: void rowPerm::resize( int r )
  516: {
  517:     if( rp!=NULL ) delete [] rp;
  518:     rp = new int[r];
  519:     nr = r;
  520: }
  521: int& rowPerm::operator[](const int index)
  522: {
  523:     //assert( index>=0 && index<nr );
  524:     if( index<0 || index>=nr )
  525:     {
  526:         fprintf(stderr,"Out of bounds rowPerm access: index=%d, vector
size=%d\n",index,nr);
  527:         exit(-1);
  528:     }
  529:     return rp[index];
  530: }
  531: sparseMatrix rowPerm::operator*(const sparseMatrix &A) const
  532: {
  533:     if( this->nr == A.m )
  534:     {
  535:         int t=0;
  536:         sparseMatrix *B = new sparseMatrix(A.m, A.n, A.nnz);
  537:
  538:         matrixList temp(A.n);
  539:         list<matrixListNode>::iterator itr;
  540:
  541:         for( int i=0; i<A.nnz; i++ )
  542:         {
  543:             if( i==A.colptr[t+1] )
  544:                 t++;
  545:             temp.insert( rp[A.adjncy[i]], t, A.nzval[i] );
  546:         }
  547:         int a=0;
  548:         B->colptr[0] = 0;
  549:         for( int i=0; i<A.n; i++ )
  550:         {
  551:             B->colptr[i+1] = B->colptr[i] + temp.listVector[i].size();
  552:             itr = temp.listVector[i].begin();
  553:             for( ; itr!=temp.listVector[ i ].end() ; itr++ )
  554:             {
  555:                 B->adjncy[a] = itr->row;
  556:                 B->nzval [a] = itr->value;
  557:                 a++;
  558:             }
  559:         }
  560:
  561:         return(*B);
  562:     }
  563:     else
```

```
  564:     {
  565:         fprintf(stderr,
  566:                 "Illegal matrix row permutation: [%dx%d]*[%dx%d]\n",
  567:                 this->nr,this->nr,
  568:                 A.m, A.n
  569:                 );
  570:         sparseMatrix *B = new sparseMatrix;
  571:         return(*B);
  572:     }
  573: }
  574:
  575: //class colPerm-----------------------------------------------------------
  576: colPerm::colPerm(int cols):nc(cols)
  577: {
  578:     cp = new int[cols];
  579: }
  580: colPerm::~colPerm()
  581: {
  582:     if( cp!=NULL ) delete[] cp;
  583: }
  584: int& colPerm::operator[](const int index)
  585: {
  586:     //assert( index>=0 && index<nc );
  587:     if( index<0 || index>=nc )
  588:     {
  589:         fprintf(stderr,"Out of bounds colPerm access: index=%d, vector
size=%d\n",index,nc);
  590:         exit(-1);
  591:     }
  592:     return cp[index];
  593: }
  594:
  595: //class sparseMatrix------------------------------------------------------
  596: sparseMatrix::sparseMatrix( int mm, int nn, int nz ):m(mm),n(nn),nnz(nz)
  597: {
  598:     nzval  = new double[nz];
  599:     adjncy = new int[nz];
  600:     colptr = new int[nn+1];
  601:     colptr[nn] = nz;
  602: }
  603: sparseMatrix::sparseMatrix(const sparseMatrix& B):
  604:     m(0),
  605:     n(0),
  606:     nnz(0),
  607:     nzval(NULL),
  608:     adjncy(NULL),
  609:     colptr(NULL)
  610: {
  611:     *this = B;
  612: }
  613: sparseMatrix::~sparseMatrix()
  614: {
  615:     clear();
  616: }
  617: void sparseMatrix::dump() const
  618: {
  619:     printf("%dx%d [%d]\n",m,n,nnz);
  620:     for( int i=0; i<nnz; i++ )
  621:         printf("%3.1lf ",nzval[i]);
  622:     printf("\n");
  623:     for( int i=0; i<nnz; i++ )
```

```
  624:         printf("%3d ",adjncy[i]);
  625:     printf("\n");
  626:     for( int i=0; i<n+1; i++ )
  627:         printf("%3d ",colptr[i]);
  628:     printf("\n");
  629: }
  630: void sparseMatrix::print(char *fmt, char *zfmt) const
  631: {
  632:     denseMatrix dense(m,n);
  633:     dense = *this;
  634:     dense.print(fmt,zfmt);
  635: }
  636: void sparseMatrix::xport() const
  637: {
  638:     denseMatrix dense(m,n);
  639:     dense = *this;
  640:     dense.xport();
  641: }
  642: char* sparseMatrix::rbError(int e)
  643: {
  644:     switch(e)
  645:     {
  646:         case RBIO_OK             : return (char*) "OK"
; break;
  647:         case RBIO_CP_INVALID     : return (char*) "column pointers are
invalid"           ; break;
  648:         case RBIO_ROW_INVALID    : return (char*) "row indices are out of
range"          ; break;
  649:         case RBIO_DUPLICATE      : return (char*) "duplicate entry"
; break;
  650:         case RBIO_EXTRANEOUS     : return (char*) "Extraneous entries in
symmetric matrix"; break; //"entries in upper triangular part of symmetric matrix"
  651:         case RBIO_TYPE_INVALID   : return (char*) "matrix type invalid"
; break;
  652:         case RBIO_DIM_INVALID    : return (char*) "matrix dimensions invalid"
; break;
  653:         case RBIO_JUMBLED        : return (char*) "matrix contains unsorted
columns"      ; break;
  654:         case RBIO_ARG_ERROR      : return (char*) "input arguments invalid"
; break;
  655:         case RBIO_OUT_OF_MEMORY  : return (char*) "out of memory"
; break;
  656:         case RBIO_MKIND_INVALID  : return (char*) "mkind is invalid"
; break;
  657:         case RBIO_UNSUPPORTED    : return (char*) "finite-element form
unsupported"       ; break;
  658:         case RBIO_HEADER_IOERROR: return (char*) "header I/O error"
; break;
  659:         case RBIO_CP_IOERROR     : return (char*) "column pointers I/O error"
; break;
  660:         case RBIO_ROW_IOERROR    : return (char*) "row indices I/O error"
; break;
  661:         case RBIO_VALUE_IOERROR  : return (char*) "numerical values I/O error"
           ; break;
  662:         default                  : return (char*) "unknown error"
; break;
  663:     }
  664: }
  665: int sparseMatrix::rbRead(char *filename)
  666: {
  667:     double xr, xz, xmin, xmax ;
  668:     double *Ax, *Az ;
  669:     long int nrow, ncol, mkind, skind, *Ap, *Ai, *Zp, *Zi, asize, mkind2,
```

```
skind2,
  670:          znz, status, njumbled, xsize, nelnz;
  671:      int BUILD_UPPER=0;   //if true, construct upper part for sym. matrices
  672:      int ZERO_HANDLING=1; //0: do nothing, 1: prune zeros, 2: extract zeros
  673:                           //Zero_handling should be 1. Was 0 before 2011-07-21
  674:      char title [73], key [9], mtype [4];
  675:      UFconfig config ;
  676:      //config.malloc_memory = malloc ;
  677:      //config.free_memory = free ;
  678:      config.malloc_memory = NULL ;  //2011-07-12: Try Defaults malloc and free
  679:      config.free_memory = NULL ;
  680:
  681:      title[0] = '\0';
  682:      status = RBread( filename, BUILD_UPPER, ZERO_HANDLING, title, key,
mtype, &nrow, &ncol,
  683:                  &mkind, &skind, &asize, &znz,
  684:                  &Ap, &Ai, &Ax, &Az, &Zp, &Zi, &config );
  685:
  686:      //Print Matrix Info-----------------------------------------
  687:      //
  688:      /*
  689:      printf("RBio status: ");
  690:      if( status < 0 )     printf("%d: Error\n",  status);
  691:      else if( status==0 ) printf("%d: Ok\n",     status);
  692:      else                 printf("%d: Warning\n",status);
  693:      printf("Matrix Size: %dx%d\n", nrow, ncol );
  694:      printf("NNZ: %d\n",asize/2);
  695:      printf("Title: %s\n", title);
  696:      printf("Key: %s\n", key);
  697:      printf("Type: %s\n", mtype);
  698:      */
  699:      //-----------------------------------------------------------
  700:
  701:      if (status != RBIO_OK)
  702:      {
  703:          fprintf (stderr,"RBread Error!  %ld  %s\n    > %s\n", status,
filename,rbError(status)) ;
  704:          UFfree(Ap, &config);
  705:          UFfree(Ai, &config);
  706:          UFfree(Ax, &config);
  707:          UFfree(Az, &config);
  708:          UFfree(Zp, &config);
  709:          UFfree(Zi, &config);
  710:          return 1;
  711:      }
  712:
  713:      this->clear();
  714:
  715:      m = nrow;
  716:      n = ncol;
  717:      nnz = Ap[ncol];
  718:
  719:      nzval  = new double[nnz];
  720:      adjncy = new int[nnz];
  721:      colptr = new int[n+1];
  722:      colptr[n] = nnz;
  723:
  724:      for( int i=0; i<nnz; i++ )
  725:      {
  726:          nzval[i] = Ax[i];
  727:          adjncy[i] = Ai[i];
  728:      }
  729:      for( int i=0; i<n; i++ )
```

```
  730:         colptr[i] = Ap[i];
  731:
  732:     UFfree(Ap, &config);
  733:     UFfree(Ai, &config);
  734:     UFfree(Ax, &config);
  735:     UFfree(Az, &config);
  736:     UFfree(Zp, &config);
  737:     UFfree(Zi, &config);
  738:     return 0;
  739: }
  740: void sparseMatrix::rbWrite(char *filename, char *title, char *key)
  741: {
  742:     long int status, nrow, ncol, *Ap=NULL, *Ai=NULL;
  743:     double *Ax=NULL;
  744:     char mtype2[4]="\0\0\0";
  745:
  746:     nrow = m;
  747:     ncol = n;
  748:
  749:     Ap = new long int[n+1];
  750:     Ai = new long int[nnz];
  751:     Ax = new double[nnz];
  752:
  753:     assert(Ap!=NULL);
  754:     assert(Ai!=NULL);
  755:     assert(Ax!=NULL);
  756:
  757:     for( int i=0; i<n+1; i++ )
  758:         Ap[i] = colptr[i];
  759:     for( int i=0; i<nnz; i++ )
  760:     {
  761:         Ax[i] = nzval[i];
  762:         Ai[i] = adjncy[i];
  763:     }
  764:
  765:     status = RBwrite (filename, title, key, nrow, ncol, Ap, Ai, Ax, NULL,
NULL, NULL,
  766:         0, mtype2, NULL) ;
  767:
  768:     delete[] Ap;
  769:     delete[] Ai;
  770:     delete[] Ax;
  771:
  772:     if(status != RBIO_OK)
  773:     {
  774:         fprintf( stderr, "RBwrite error!  %ld  %s\n    > %s\n", status,
basename(filename), rbError(status) );
  775:     }
  776: }
  777:
  778: double sparseMatrix::at( int row, int col  ) const
  779: {
  780:     double val=0;
  781:     for( int i=this->colptr[col]; i<this->colptr[col+1] && this
->adjncy[i]<=row; i++ )
  782:     {
  783:         if( this->adjncy[i]==row )
  784:             val = this->nzval[i];
  785:     }
  786:     return val;
  787: }
  788: void sparseMatrix::check()
  789: {
```

```
  790:     for( int i=0; i<n; i++ )
  791:     {
  792:         if( colptr[i]>=colptr[i+1] )
  793:             printf("colptr[%d] = %d  colptr[%d] = %d\n"
,i,colptr[i],i+1,colptr[i+1]);
  794:     }
  795: }
  796: int sparseMatrix::dz()
  797: {
  798:     int count=0;
  799:     for( int i=0; i<m; i++ )
  800:     {
  801:         if( i>=n )
  802:             break;
  803:         if( at(i,i)==0 )
  804:             count++;
  805:     }
  806:     return count;
  807: }
  808: int sparseMatrix::d1()
  809: {
  810:     int count=0;
  811:     for( int i=0; i<m; i++ )
  812:     {
  813:         if( i>=n )
  814:             break;
  815:         if( at(i,i)==1 )
  816:             count++;
  817:     }
  818:     return count;
  819: }
  820: int sparseMatrix::emptyCols()
  821: {
  822:     int count=0;
  823:     for( int i=0; i<n; i++ )
  824:         if( colptr[i]<0 )
  825:             count++;
  826:     return count;
  827: }
  828:
  829: sparseMatrix sparseMatrix::operator*(colPerm &C) const
  830: {
  831:     if( this->n == C.nc )
  832:     {
  833:         int k=0;
  834:         sparseMatrix *B = new sparseMatrix(this->m, this->n, this->nnz);
  835:         for( int i=0; i<C.nc; i++ )
  836:         {
  837:             B->colptr[i] = k;
  838:             for( int j=this->colptr[C[i]]; j<this->colptr[C[i]+1]; j++ )
  839:             {
  840:                 B->nzval[k] = this->nzval[j];
  841:                 B->adjncy[k]= this->adjncy[j];
  842:                 k++;
  843:             }
  844:         }
  845:         return(*B);
  846:     }
  847:     else
  848:     {
  849:         fprintf(stderr,
  850:                 "Illegal matrix column permutation: [%dx%d]*[%dx%d]\n",
  851:                 this->m,this->n,
```

```
  852:                 C.nc, C.nc
  853:                 );
  854:         sparseMatrix *B = new sparseMatrix;
  855:         return(*B);
  856:     }
  857: }
  858: sparseMatrix sparseMatrix::operator*( const sparseMatrix &B )
  859: {
  860:     /*
  861:         M*N  M*N  => N==M and C == M*N
  862:         | |__| |
  863:         |______|
  864:     */
  865:
  866:     if( n!=B.m )
  867:     {
  868:         fprintf( stderr, "Invalid dimensions: A: %dx%d, B: %dx%d\n", m, n,
B.m, B.n );
  869:         abort();
  870:     }
  871:
  872:     CSFBlock *C = new CSFBlock(m, B.n);
  873:     assert(C!=NULL);
  874:
  875:     //Create a temporary vector for column in B
  876:     //
  877:     double *column = new double [B.m];
  878:     assert( column!=NULL );
  879:     for( int i=0; i<B.m; i++ )
  880:         column[i] = 0;
  881:
  882:     //Convert A to CSR
  883:     //
  884:     CSFBlock *csrA = new CSFBlock( m, n );
  885:     assert(csrA!=NULL);
  886:     *csrA = *this; //Convert A to CSFBlock
  887:     *csrA = csrA->toCSR();
  888:
  889:
  890:     // Iterate through columns of B
  891:     //
  892:     int jB=0;
  893:     for( int i=0; i<B.n; i++ )
  894:     {
  895:         if( B.colptr[i]<0 )
  896:             continue;
  897:
  898:         jB=i;
  899:         while( B.colptr[++jB]<0 );
  900:
  901:
  902:         //1. Copy elements from B to dense column
  903:         //
  904:         for( int k=B.colptr[i]; k<B.colptr[jB]; k++ )
  905:             column[B.adjncy[k]] = B.nzval[k];
  906:
  907:
  908:         //2. Iterate through rows in A
  909:         //
  910:         for( int r=0; r<m; r++ )
  911:         {
  912:             double product=0;
  913:             int jA=r;
```

```
914:             while( csrA->P[++jA]<0 );
915:             for( int k=csrA->P[r]; k<csrA->P[jA]; k++ )
916:                 product += csrA->V[k]*column[ csrA->I[k] ];
917:             C->insertCSC(r,i,product);
918:         }
919:
920:
921:         //3. Clear dense column
922:         //
923:         for( int k=B.colptr[i]; k<B.colptr[jB]; k++ )
924:             column[B.adjncy[k]] = 0;
925:     }
926:
927:
928:     sparseMatrix *R = new sparseMatrix;
929:
930:     (*R) = (*C);
931:
932:     delete csrA;
933:     delete [] column;
934:     delete C;
935:
936:     return (*R);
937: }
938: denseMatrix sparseMatrix::operator*( const denseMatrix &B )
939: {
940:     /*
941:         M*N  M*N  => N==M and C == M*N
942:         |  |__|  |
943:         |_______|
944:     */
945:
946:     if( static_cast<unsigned int>(n)!=B.m )
947:     {
948:         fprintf( stderr, "Invalid dimensions: A: %dx%d, B: %dx%d\n", m, n,
B.m, B.n );
949:         abort();
950:     }
951:
952:
953:     denseMatrix R(m, B.n);
954:
955:     //Convert A to CSR
956:     //
957:     CSFBlock *csrA = new CSFBlock( m, n );
958:     assert(csrA!=NULL);
959:     *csrA = *this; //Convert A to CSFBlock
960:     *csrA = csrA->toCSR();
961:
962:     for( unsigned int i=0; i<B.n; i++ )
963:     {
964:         //2. Iterate through rows in A
965:         //
966:         for( int r=0; r<m; r++ )
967:         {
968:             double product=0;
969:             int jA=r;
970:             while( csrA->P[++jA]<0 );
971:             for( int k=csrA->P[r]; k<csrA->P[jA]; k++ )
972:                 product += csrA->V[k]*B[ csrA->I[k] ][i];
973:
974:             R[r][i] = product;
975:         }
```

```
  976:     }
  977:
  978:     delete csrA;
  979:
  980:     return R;
  981: }
  982: void sparseMatrix::operator=(const sparseMatrix &B)
  983: {
  984:     if( this!=&B )
  985:     {
  986:         if( this->m != B.m || this->n != B.n || this->nnz != B.nnz )
  987:         {
  988:             this->clear();
  989:             this->m = B.m;
  990:             this->n = B.n;
  991:             this->nnz=B.nnz;
  992:             this->nzval  = new double[this->nnz];
  993:             this->adjncy = new int[this->nnz];
  994:             this->colptr = new int[this->n+1];
  995:             this->colptr[this->n] = this->nnz;
  996:         }
  997:         for( int i=0; i<nnz; i++ )
  998:         {
  999:             this->nzval[i] = B.nzval[i];
 1000:             this->adjncy[i] = B.adjncy[i];
 1001:         }
 1002:         for( int i=0; i<n; i++ )
 1003:             this->colptr[i] = B.colptr[i];
 1004:     }
 1005: }
 1006: void sparseMatrix::operator=(const denseMatrix &B)
 1007: {
 1008:     this->clear();
 1009:     m = B.m;
 1010:     n = B.n;
 1011:
 1012:     nnz=0;
 1013:     for( unsigned int i=0; i<B.m; i++ )
 1014:     for( unsigned int j=0; j<B.n; j++ )
 1015:         if( B[i][j]!=0 )
 1016:             nnz++;
 1017:
 1018:     nzval = new double[nnz];
 1019:     adjncy = new int[nnz];
 1020:     colptr = new int[B.n+1];
 1021:     colptr[B.n] = nnz;
 1022:
 1023:     unsigned int i=0;
 1024:     int k=0;
 1025:     int c=0;
 1026:     for( unsigned int j=0; j<B.n; j++ )
 1027:     {
 1028:         c=0;
 1029:         for( i=0; i<B.m; i++ )
 1030:             if( B[i][j]!=0 )
 1031:             {
 1032:                 adjncy[k] = i;
 1033:                 if(c==0) colptr[j] = k;
 1034:                 nzval[k++] = B[i][j];
 1035:                 c=1;
 1036:             }
 1037:     }
 1038: }
```

```
1039: void sparseMatrix::operator=(const CSFBlock &B)
1040: {
1041:     this->clear();
1042:     m = B.m;
1043:     n = B.n;
1044:     nnz = B.P[n];
1045:
1046:     nzval = new double[nnz];
1047:     adjncy = new int[nnz];
1048:     colptr = new int[n+1];
1049:     colptr[n] = nnz;
1050:
1051:     for( int i=0; i<nnz; i++ )
1052:     {
1053:         nzval[i] = B.V[i];
1054:         adjncy[i] = B.I[i];
1055:     }
1056:     for( int i=0; i<n; i++ )
1057:         colptr[i] = B.P[i];
1058: }
1059: void sparseMatrix::clear()
1060: {
1061:     if( nzval !=NULL ){
1062:         delete[] nzval;
1063:         nzval = NULL;
1064:     }
1065:     if( adjncy!=NULL ){
1066:         delete[] adjncy;
1067:         adjncy = NULL;
1068:     }
1069:     if( colptr!=NULL ){
1070:         delete[] colptr;
1071:         colptr = NULL;
1072:     }
1073:     m   = 0;
1074:     n   = 0;
1075:     nnz = 0;
1076: }
1077: //class
denseMatrix---------------------------------------------------------------
1078: denseMatrix::denseMatrix():
1079:     m(0),
1080:     n(0),
1081:     _Array(NULL)
1082: {}
1083: denseMatrix::denseMatrix( unsigned int s ):
1084:     m(s),
1085:     n(s),
1086:     _Array( new double[s*s] )
1087: {
1088:     for(unsigned int i=0; i<s*s; i++)
1089:         _Array[i] = 0;
1090: }
1091: denseMatrix::denseMatrix( unsigned int mm, unsigned int nn ):
1092:     m(mm),
1093:     n(nn),
1094:     _Array( new double[mm*nn] )
1095: {
1096:     for(unsigned int i=0; i<m*n; i++)
1097:         _Array[i] = 0;
1098: }
1099: denseMatrix::~denseMatrix()
1100: {
```

```
1101:     delete [] _Array;
1102: }
1103: double* denseMatrix::operator[](unsigned int Row)
1104: {
1105:     return _Array + Row*n;
1106: }
1107: const double* denseMatrix::operator[](unsigned int Row) const
1108: {
1109:     return _Array + Row*n;
1110: }
1111: void denseMatrix::clear()
1112: {
1113:     for( unsigned int i=0; i<m*n; i++ )
1114:         _Array[i]=0;
1115: }
1116: bool denseMatrix::hasNNZ()
1117: {
1118:     for( unsigned int i=0; i<m*n; i++ )
1119:         if( _Array[i]!=0 )
1120:             return true;
1121:     return false;
1122: }
1123: unsigned int denseMatrix::nnz() const
1124: {
1125:     unsigned int count = 0;
1126:     for( unsigned int i=0; i<m*n; i++ )
1127:         if( _Array[i]!=0 )
1128:             count++;
1129:     return count;
1130: }
1131: void denseMatrix::fromCSR(const CSFBlock *csr)
1132: {
1133:     int row=0, j=0;
1134:     while( csr->P[row]<0 ) row++;
1135:     j=row;
1136:     while( csr->P[++j]<0 );
1137:     for( int i=0; i<csr->P[m]; i++ )
1138:     {
1139:         if( i==csr->P[j] )
1140:         {
1141:             row = j;
1142:             while( csr->P[++j]<0 );
1143:         }
1144:         (*this)[row][csr->I[i]] = csr->V[i];
1145:     }
1146: }
1147: void denseMatrix::fromCSC(const CSFBlock *csc)
1148: {
1149:     int col=0, j=0;
1150:     while( csc->P[++j]<0 );
1151:     for( int i=0; i<csc->P[n]; i++ )
1152:     {
1153:         if( i==csc->P[j] )
1154:         {
1155:             col = j;
1156:             while( csc->P[++j]<0 );
1157:         }
1158:         (*this)[csc->I[i]][col] = csc->V[i];
1159:     }
1160: }
1161: void denseMatrix::print(char *fmt, char *zfmt) const
1162: {
1163:     for( unsigned int i=0; i<m*n; i++ )
```

```
 1164:     {
 1165:         if( _Array[i]==0 )
 1166:             printf(zfmt, 0);
 1167:         else
 1168:             printf(fmt, _Array[i]);
 1169:         if( (i+1)%n==0 )
 1170:             printf("\n");
 1171:     }
 1172: }
 1173: int denseMatrix::import( char *filename )
 1174: {
 1175:     FILE *infile = NULL;
 1176:     infile = fopen( filename, "r" );
 1177:     if( !infile )
 1178:         return 1;
 1179:
 1180:     if( fscanf( infile, ";Size: %ux%u\n", &m, &n ) != 2 )
 1181:         return 2;
 1182:
 1183:     if( _Array!=NULL )
 1184:         delete [] _Array;
 1185:     _Array = new double[m*n];
 1186:     assert(_Array!=NULL );
 1187:
 1188:     for( unsigned int i=0; i<m*n; i++ )
 1189:         _Array[i] = 0;
 1190:
 1191:     fscanf( infile, "%*[^\n]\n" );
 1192:
 1193:     for( unsigned int i=0; i<m*n; i++ )
 1194:         if( fscanf( infile, "%lf%*[,;]", _Array+i )!=1 )
 1195:             return 3;
 1196:
 1197:     fclose(infile);
 1198:     return 0;
 1199: }
 1200: void denseMatrix::xport(char *filename) const
 1201: {
 1202:     FILE *outfile = NULL;
 1203:
 1204:     if( !filename )
 1205:         outfile = stdout;
 1206:     else
 1207:         outfile = fopen( filename, "w" );
 1208:
 1209:     if( !outfile )
 1210:     {
 1211:         if( filename!=NULL )
 1212:             fprintf( stderr, "Could not write to output file: %s\n",
filename );
 1213:         return;
 1214:     }
 1215:
 1216:     fprintf(outfile,";Size: %dx%d\n", m, n );
 1217:     fprintf(outfile,"=[\n");
 1218:     for( unsigned int i=0; i<m*n; i++ )
 1219:     {
 1220:         if( _Array[i]==0 )
 1221:             fprintf(outfile,"%22d", 0);
 1222:         else
 1223:             fprintf(outfile,"%22.16e", _Array[i]);
 1224:
 1225:         if( (i+1)%n==0 )
```

```
1226:             fprintf(outfile,";\n");
1227:         else
1228:             fprintf(outfile,", ");
1229:     }
1230:     fprintf(outfile,"]\n");
1231: 
1232:     if( filename!=NULL )
1233:         fclose( outfile );
1234: }
1235: void denseMatrix::operator=(const sparseMatrix &B)
1236: {
1237:     if(m!=(unsigned int)B.m && n!=(unsigned int)B.n)
1238:     {
1239:         m = B.m;
1240:         n = B.n;
1241:         delete [] _Array;
1242:         _Array = new double[m*n];
1243:         assert(_Array!=NULL);
1244:     }
1245: 
1246:     for( unsigned int j=0; j<m*n; j++ )
1247:         _Array[j]=0;
1248: 
1249:     int i=0;
1250:     for( int k=0; k<B.nnz; k++ )
1251:     {
1252:         if( k==B.colptr[i+1] ) i++;
1253:         (*this)[ B.adjncy[k] ][i] = B.nzval[k];
1254:     }
1255: }
1256: denseMatrix denseMatrix::operator*(const denseMatrix &B) const
1257: {
1258:     if( n == B.m )
1259:     {
1260:         denseMatrix *C = new denseMatrix( m, B.n );
1261:         for( unsigned int i=0; i<C->m; i++ )
1262:         for( unsigned int j=0; j<C->n; j++ )
1263:         for( unsigned int k=0; k<n; k++ )
1264:             (*C)[i][j] += (*this)[i][k] * B[k][j];
1265:         return(*C);
1266:     }
1267:     else
1268:     {
1269:         fprintf(stderr,
1270:                 "Illegal matrix operation: [%dx%d]*[%dx%d]\n",
1271:                 m,n,
1272:                 B.m, B.n
1273:                 );
1274:         denseMatrix *C = new denseMatrix;
1275:         return *C;
1276:     }
1277: }
1278: void denseMatrix::operator=(const denseMatrix &B)
1279: {
1280:     if( B.m==0 || B.n==0 )
1281:     {
1282:         m=0;
1283:         n=0;
1284:         if(_Array!=NULL)
1285:         {
1286:             delete [] _Array;
1287:             _Array = NULL;
1288:         }
```

```
1289:         return;
1290:     }
1291:     if( m != B.m || n != B.n )
1292:     {
1293:         m = B.m;
1294:         n = B.n;
1295:         if(_Array!=NULL)
1296:         {
1297:             delete [] _Array;
1298:             _Array = NULL;
1299:         }
1300:         _Array = new double[m*n];
1301:         assert(_Array!=NULL);
1302:     }
1303:     for( unsigned int i=0; i<m; i++ )
1304:     for( unsigned int j=0; j<n; j++ )
1305:         (*this)[i][j] = B[i][j];
1306: }
1307:
1308: //class hypergraph-----------------------------------------------------------
1309:
1310: hypergraph::hypergraph():
1311: nets  (NULL),
1312: vertx (NULL),
1313: xpins (NULL),
1314: pins  (NULL),
1315: c(0),
1316: n(0)
1317: {};
1318:
1319: hypergraph::~hypergraph()
1320: {
1321:     if(  nets!=NULL  ) delete [] nets;
1322:     if( vertx!=NULL  ) delete [] vertx;
1323:     if( xpins!=NULL  ) delete [] xpins;
1324:     if(  pins!=NULL  ) delete [] pins;
1325: }
1326:
1327: void hypergraph::print() const
1328: {
1329:     printf("  c: %d\n",c);
1330:     printf("  n: %d\n",n);
1331:
1332:     //Print Cells/Vertices
1333:     printf("  C[]: ");
1334:     for( int i=0; i<c; i++ )
1335:         printf("%2d ",vertx[i]);
1336:     printf("\n");
1337:
1338:     //Print nets
1339:     printf("  N[]: ");
1340:     int j=0;
1341:     for( int i=0; i<xpins[n]; i++ )
1342:         if( i==xpins[j] )
1343:             printf("%2d ",nets[j++]);
1344:         else printf("   ");
1345:     printf("\n");
1346:
1347:     //Print xpins[]
1348:     printf("  X[]: ");
1349:     j=0;
1350:     for( int i=0; i<=xpins[n]; i++ )
```

```
1351:         if( i==xpins[j] )
1352:             printf("%2d ",xpins[j++]);
1353:         else printf("   ");
1354:     printf("\n");
1355:
1356:     //Print pins[]
1357:     printf("  P[]: ");
1358:     for( int i=0; i<xpins[n]; i++ )
1359:         printf("%2d ",pins[i]);
1360:     printf("\n");
1361: }
1362:
1363: //Converts a matrix to a column-net hypergraph
1364: void hypergraph::operator=(const denseMatrix &M)
1365: {
1366:     c = M.m;     //Number of cells (vertices)
1367:     n = M.n;     //Number of nets (columns)
1368:     nets  = new int[n];
1369:     vertx = new int[c];
1370:     xpins = new int[n+1];
1371:
1372:     list<int> pinList;
1373:
1374:     for( int i=0; i<c; i++ )
1375:         vertx[i] =  i;
1376:     for( unsigned int i=0; i<M.n; i++ )
1377:     {
1378:         nets[i]  = i;
1379:         xpins[i] = pinList.size();
1380:         for( unsigned int j=0; j<M.m; j++ )
1381:             if( M[j][i]!=0 )
1382:                 pinList.push_back(j);
1383:     }
1384:     xpins[n] = pinList.size();
1385:     this->pins   = new int[ pinList.size() ];
1386:     int j=0;
1387:     for( list<int>::iterator i=pinList.begin(); i!=pinList.end(); i++ )
1388:         this->pins[j++] = *i;
1389: }
1390:
1391: void hypergraph::operator=(const sparseMatrix &M)
1392: {
1393:     c = M.m;
1394:     n = M.n;
1395:     nets  = new int[n];
1396:     vertx = new int[c];
1397:     xpins = new int[n+1];
1398:     pins  = new int[M.nnz];
1399:
1400:     xpins[n] = M.nnz;
1401:
1402:     for( int i=0; i<c; i++ )
1403:         vertx[i] =  i;
1404:     for( int i=0; i<n; i++ )
1405:     {
1406:         nets[i] = i;
1407:         xpins[i] = M.colptr[i];
1408:     }
1409:     for( int i=0; i<M.nnz; i++ )
1410:         pins[i] = M.adjncy[i];
1411: }
1412:
1413: //Class
```

```
matrixList-----------------------------------------------------------------
 1414: matrixList::matrixList(int s):listVector(s)
 1415: {}
 1416: void matrixList::insert(unsigned int r, unsigned int c, double v)
 1417: {
 1418:     if( c >= listVector.size() || v==0 )
 1419:         return;
 1420:     list<matrixListNode>::iterator itr;
 1421:     itr = listVector[ c ].begin();
 1422:     for( ; itr!=listVector[ c ].end() ; itr++ )
 1423:     {
 1424:         if( itr->row == r )
 1425:         {
 1426:             fprintf(stderr,"matrixList warning: Duplicate value at
[%d][%d]!\n",r,c);
 1427:             return;
 1428:         }
 1429:         if( itr->row > r )
 1430:             break;
 1431:     }
 1432:     listVector[ c ].insert(itr,matrixListNode( r, v ));
 1433: }
 1434: unsigned int matrixList::size()
 1435: {
 1436:     unsigned int count = 0;
 1437:     for( unsigned int i=0; i<listVector.size(); i++ )
 1438:         count += listVector[i].size();
 1439:     return count;
 1440: }
 1441:
 1442: //Class
matrixVector---------------------------------------------------------------
 1443: void matrixVector::insert(unsigned int r, unsigned int c, double v)
 1444: {
 1445:     if( v==0 )
 1446:         return;
 1447:
 1448:     vectorList.push_back( matrixVectorNode(r,c,v) );
 1449: }
 1450: unsigned int matrixVector::size()
 1451: {
 1452:     return vectorList.size();
 1453: }
```

```
 1: /*
 2:     Author:  Esteban Torres
 3:     Date:    July 2010
 4:     File:    datastructures.h
 5:     Purpose: This file contains data structure definitions.
 6: */
 7:
 8: #ifndef DATASTRUCTURES_H
 9: #define DATASTRUCTURES_H
10:
11: #include <cmath>
12: #include <cstdio>
13: #include <cstddef>
14: #include <cstdlib>
15: #include <cassert>
16: #include <vector>
17: #include <list>
18: using namespace std;
19:
20: #include "UFconfig.h" //Makefile: -I$(DIR_SUITESPARSE)/UFconfig
21: #include "RBio.h"     //Makefile: -I$(DIR_SUITESPARSE)/RBio/Include
22:
23: class sparseMatrix;
24: class denseMatrix;
25:
26: class CSFBlock //Compact Storage Format Block
27: {
28:     private:
29:         int currentIndex;
30:         int currentRow;
31:         int currentCol;
32:
33:     public:
34:         int m;
35:         int n;
36:         vector<double> V;  //Values
37:         vector<int> I;     //Indices
38:         vector<int> P;     //Pointers
39:
40:         CSFBlock(int mm, int nn);
41:         ~CSFBlock();
42:
43:         void operator=( const CSFBlock &B );
44:         void operator=( const sparseMatrix &B );
45:         bool operator!=( const CSFBlock &B) const;
46:         CSFBlock toCSR();
47:
48:         void reserve( int size );
49:
50:         bool isEnd();
51:         double atCSR( int row, int col );
52:         double atCSC( int row, int col );
53:
54:         void insertCSR(int row, int col, double value);
55:         void insertCSC(int row, int col, double value);
56:         void getFirstCSC( int &r, int &c, double &v );
57:         void getFirstCSR( int &r, int &c, double &v );
58:         void getCurrent(  int &r, int &c, double &v );
59:         bool getNextCSC( int &r, int &c, double &v );
60:         bool getNextCSR( int &r, int &c, double &v );
61:         double getDiagonal( int i );
62:
63:         void fromDenseCSC(const denseMatrix &csc);
```

```
    64:         void fromDenseCSR(const denseMatrix &csr);
    65:
    66:         void clear();
    67:         void dump() const;
    68:         void printCSC() const;
    69:         void printCSR() const;
    70:         int nnz() const;
    71:
    72:         void check();
    73: };
    74:
    75:
    76: class rowPerm
    77: {
    78:     private:
    79:         int *rp; //Row Permutation Vector
    80:     public:
    81:         int nr;  //Number of Rows
    82:         rowPerm():nr(0),rp(NULL){};
    83:         rowPerm(int rows);
    84:         ~rowPerm();
    85:
    86:         //This allows: sparseMatrix = rowPerm * sparseMatrix;
    87:         sparseMatrix operator*(const sparseMatrix &A) const;
    88:         void resize(int r);
    89:
    90:         //Overload the subscript operator
    91:         int& operator[](const int index);
    92: };
    93:
    94: class colPerm
    95: {
    96:     private:
    97:         int *cp; //Column Permutation Vector
    98:     public:
    99:         int nc;  //Number of columns
   100:         colPerm():nc(0),cp(NULL){};
   101:         colPerm(int cols);
   102:         ~colPerm();
   103:
   104:         //Overload the subscript operator
   105:         int& operator[](const int index);
   106: };
   107:
   108:
   109: class sparseMatrix
   110: {
   111:     public:
   112:         int m;
   113:         int n;
   114:         int nnz;
   115:         double *nzval;
   116:         int *adjncy;
   117:         int *colptr;
   118:         sparseMatrix():m(0),n(0),nnz(0),
   119:                        nzval(NULL),adjncy(NULL),colptr(NULL){};
   120:         sparseMatrix( int mm, int nn, int nz );
   121:         sparseMatrix( const sparseMatrix& B );
   122:         ~sparseMatrix();
   123:         void dump() const;
   124:         //void print(char *fmt=(char*)"%4.1lf ", char *zfmt=(char*)"   0 ")
const;
   125:         void print(char *fmt=(char*)"%8.1e ", char *zfmt=(char*)"%8d ") const
```

```
;
  126:          void xport() const;
  127:          char* rbError(int e);
  128:          int  rbRead(char*);
  129:          void rbWrite(char *filename, char *title=(char*)"", char *key=(char*)
"");
  130:          double at( int row, int col  ) const; //return the value of element
at [row,col]
  131:          void check();
  132:          int dz();
  133:          int d1();
  134:          int emptyCols();
  135:
  136:          //This allows: sparseMatrix = sparseMatrix * colPerm;
  137:          sparseMatrix operator*(colPerm &C) const;
  138:          sparseMatrix operator*(const sparseMatrix &B);
  139:          denseMatrix operator*(const denseMatrix &B);
  140:          void operator=(const sparseMatrix &B);
  141:          void operator=(const denseMatrix &B);
  142:          void operator=(const CSFBlock &B);
  143:          void clear();
  144: };
  145:
  146: class denseMatrix
  147: {
  148:     private:
  149:          double* _Array;
  150:     public:
  151:          unsigned int m;
  152:          unsigned int n;
  153:          denseMatrix();
  154:          denseMatrix( unsigned int s );
  155:          denseMatrix( unsigned int mm, unsigned int nn );
  156:          ~denseMatrix();
  157:
  158:          double* operator[](unsigned int Row);
  159:          const double* operator[](unsigned int Row) const;
  160:
  161:          void clear();
  162:          bool hasNNZ();
  163:          unsigned int nnz() const;
  164:          void fromCSR(const CSFBlock *csr);
  165:          void fromCSC(const CSFBlock *csc);
  166:
  167:          void print(char *fmt=(char*)"%8.1e ", char *zfmt=(char*)"%8d ") const
;
  168:          int import(char *filename);
  169:          void xport(char *filename=NULL) const;
  170:          void operator=(const sparseMatrix &B);
  171:          denseMatrix operator*(const denseMatrix &B) const;
  172:          void operator=(const denseMatrix &B);
  173:
  174:          double* getPointer(){return _Array;};
  175: };
  176:
  177: class hypergraph
  178: {
  179:     public:
  180:          int * nets;  //Net IDs
  181:          int *vertx;  //Vertex IDs
  182:          int *xpins;  //Pin indexes for each net
  183:          int * pins;  //Vertex IDs for each edge
  184:          int c;       //number of cells/vertices
```

```
  185:         int n;       //number of nets
  186:
  187:         hypergraph();
  188:         ~hypergraph();
  189:         void print() const;
  190:         void operator=(const denseMatrix &M);
  191:         void operator=(const sparseMatrix &M);
  192: };
  193:
  194: class matrixListNode
  195: {
  196:     public:
  197:         unsigned int row;
  198:         double value;
  199:         matrixListNode(unsigned int r, double v):row(r),value(v){};
  200: };
  201: class matrixList
  202: {
  203:     public:
  204:         matrixList(int s);
  205:         vector< list< matrixListNode > > listVector;
  206:         void insert(unsigned int r, unsigned int c, double v);
  207:         unsigned int size();
  208: };
  209:
  210:
  211: // This structure is converted to matrixList then sparseMatrix
  212: //
  213: class matrixVectorNode
  214: {
  215:     public:
  216:         unsigned int row;
  217:         unsigned int col;
  218:         double value;
  219:         matrixVectorNode(unsigned r,unsigned c,double
v):row(r),col(c),value(v){};
  220: };
  221: class matrixVector
  222: {
  223:     public:
  224:         list< matrixVectorNode > vectorList;
  225:         void insert(unsigned int r, unsigned int c, double v);
  226:         unsigned int size();
  227: };
  228:
  229: #endif
  230:
```

```
 1: /*
 2:     Author:  Esteban Torres
 3:     Date:    August 2010
 4:     File:    hund.h
 5:     Purpose: This file contains the function definitions for
 6:              the HUND reordering algorithm.
 7: */
 8:
 9: #ifndef HUND_H
10: #define HUND_H
11:
12: int MIN_BLOCK;
13:
14: #include <cstdio>
15: #include "datastructures.h"
16:
17: #include "patoh.h"     //Makefile: -I$(DIR_PATOH)
18: #include "slu_ddefs.h" //Makefile: -I$(DIR_SUPERLU)/SRC
19: #include "ccolamd.h"   //Makefile: -I$(DIR_SUITESPARSE)/CCOLAMD/Include
20:
21: #include <algorithm> //for min
22: using namespace std;
23:
24: void mc64(const sparseMatrix &A, rowPerm &P)
25: {
26:     int perm[A.n]; for( int i=0; i<A.n; i++ ) perm[i]=0;
27:     double u[A.n]; for( int i=0; i<A.n; i++ ) u[i]=0;
28:     double v[A.n]; for( int i=0; i<A.n; i++ ) v[i]=0;
29:
30:     dldperm( 5, A.n, A.nnz, A.colptr, A.adjncy, A.nzval, perm, u, v );
31:
32:     int col=0;
33:     for( int i=0; i<A.nnz; i++ )
34:     {
35:         if( i==A.colptr[col+1] )
36:             col++;
37:         A.nzval[i] = A.nzval[i] * exp( u[A.adjncy[i]] + v[col] );
38:     }
39:
40:     for( int j=0; j<A.m; j++ )
41:         P[j] = perm[j];
42: }
43:
44: void cColamd(const sparseMatrix &A, rowPerm &P, colPerm &Q, int* constraints)
45: {
46:     int ALEN = ccolamd_recommended(A.nnz, A.m, A.n);
47:
48:     int* Atemp = new int[ALEN];
49:     assert(Atemp!=NULL);
50:
51:     int* p = new int[A.n+1];
52:     int* cmember = new int[A.n]; //CCOLAMD Contraints
53:     double knobs[CCOLAMD_KNOBS]; //CCOLAMD Arguments
54:
55:     //Set CCOLAMD Arguments
56:     ccolamd_set_defaults(knobs);
57:
58:     knobs[CCOLAMD_LU] = 1; //NONZERO=TRUE, Optimize for LU Factorization
59:
60:     {
61:         int set=0;
62:         for( int i=0; i<A.n; i++ )
63:         {
```

```
    64:             cmember[i] = set;
    65:             if( constraints[i]==1 )
    66:                 set++;
    67:         }
    68:     }
    69:
    70:     for( int i=0; i<ALEN; i++ )
    71:         Atemp[i]=0;
    72:     for( int i=0; i<A.n+1; i++ )
    73:         p[i]=0;
    74:
    75:     for( int i=0; i<A.nnz; i++ )
    76:         Atemp[i] = A.adjncy[i];
    77:     for( int i=0; i<A.n+1; i++ )
    78:         p[i] = A.colptr[i];
    79:
    80:     int ok;
    81:     int stats[CCOLAMD_STATS];  //for ccolamd output statistics
    82:
    83:     //CCOLAMD Arguments
    84:     //1 int n_row;         Input: # of Rows
    85:     //2 int n_col;         Input: # of Columns
    86:     //3 int Alen;          Input: Size of working memory
    87:     //4 int A[Alen];       Input: Row indices, Note: undefined on output.
    88:     //5 int p[n_col+1];    Both input and output:
    89:     //                      IN: Column Pointers,
    90:     //                     OUT: Permutation Vector
    91:     //6 double knobs [CCOLAMD_KNOBS]; Input: CCOLAMD Parameters
    92:     //7 int stats [CCOLAMD_STATS];    Output: Statistics
    93:     //8 int cmember [n_col];          Input: Constraints
    94:     //   ccolamd (1  , 2  , 3   , 4     , 5, 6    , 7    , 8   );
    95:     ok = ccolamd (A.m, A.n, ALEN, Atemp , p, knobs, stats, cmember);
    96:
    97:     //ccolamd_report (stats) ;
    98:
    99:     if(!ok)
   100:     {
   101:         printf ("ccolamd error!\n");
   102:         exit (1) ;
   103:     }
   104:
   105:     for( int j=0; j<A.m; j++ )
   106:     {
   107:         //Q[j] = p[j];
   108:         //P[p[j]] = j;
   109:
   110:         // Changed from above version on August 21, 2012
   111:         //
   112:         Q[p[j]] = j;
   113:         P[j] = p[j];
   114:     }
   115:
   116:     delete[] Atemp;
   117:     delete[] p;
   118:     delete[] cmember;
   119: }
   120:
   121: //Input:  Hypergraph H
   122: //Output: rowPerm P, colPerm Q
   123: void hund(
   124:     const hypergraph &H, rowPerm &P, colPerm &Q,
   125:     int* constraints, int K_PARTS,
   126:     int partLevel=0, int partIndex=0,
```

```
  127:      int nNets=0, int nCells=0 )
  128: {
  129:      //
**************************************************************************
  130:      // If we have reached the minimum block size, there is no need to
continue
  131:      // recursive partitioning. Instead, compute P and Q.
  132:      //
**************************************************************************
  133:      if( H.c <= MIN_BLOCK || H.c < K_PARTS ) //Changed Nov. 8, 2010.
  134:      {
  135:          //Compute P: Row Permutation from Cells
  136:          for( int i=0; i<H.c; i++ )
  137:              P[H.vertex[i]] = nCells++;
  138:
  139:          //Compute Q: Column Permutation from Nets
  140:          for( int i=0; i<H.n; i++ )
  141:              Q[nNets++] = H.nets[i];
  142:          return;
  143:      }
  144:
  145:      PaToH_Parameters args;
  146:      int *partvec=NULL, *partweights=NULL, cut;
  147:
  148:      //Initialize parameters for partitioning
  149:      PaToH_Initialize_Parameters( &args, PATOH_CUTPART,
PATOH_SUGPARAM_DEFAULT );
  150:      args._k = K_PARTS; //Cut into K_PARTS
  151:      args.seed = 42;
  152:
  153:      //Allocate the memory that will be used for the partitioning
  154:      partvec = new int[H.c];
  155:      partweights = new int[args._k];
  156:
  157:      PaToH_Alloc( &args, H.c, H.n, 0, NULL, NULL, H.xpins, H.pins );
  158:
  159:      //   *************
  160:      //   * Partition *
  161:      //   *************
  162:      PaToH_Part( &args, H.c, H.n, 0, 0, NULL, NULL, H.xpins, H.pins,
  163:                  NULL, partvec, partweights, &cut );
  164:
  165:      PaToH_Free(); //Release memory
  166:
  167:      //
**************************************************************************
  168:      // PaToH_Part determines which cells (vertices) belong in which
partition.
  169:      // It places this information in the partvec array.
  170:      // It also reports how many NETS were cut by updating the integer 'cut'.
  171:      //
  172:      // Next, we need to determine:
  173:      //   netparts  - Which nets belong in which partition.
  174:      //   netcounts - The number of nets in each partition.
  175:      //
**************************************************************************
  176:
  177:      int netcount=0; //---------------Iterator for netparts and xpins
  178:
  179:      int* netcounts = new int[K_PARTS]; //------Number of nets in each
partition
  180:      int* edgecount = new int[K_PARTS]; //------Number of edges in each
partition
```

```
  181:     int* count = new int[K_PARTS]; //----------Number of edges connected to
each partition
  182:     int* netparts = new int[H.c]; //-----------part vector for nets
  183:
  184:     assert(netcounts!=NULL);
  185:     assert(edgecount!=NULL);
  186:     assert(count!=NULL);
  187:     assert(netparts!=NULL);
  188:
  189:
  190:     for( int i=0; i<K_PARTS; i++ ){
  191:         netcounts[i] = 0;
  192:         edgecount[i] = 0;
  193:     }
  194:
  195:     for( int i=0; i<H.xpins[H.n]; i++ ) //Iterate through H.pins
  196:     {
  197:         //Reset the counters for each net.
  198:         if( i==H.xpins[netcount] )
  199:             for( int j=0; j<K_PARTS; j++ )
  200:                 count[j] = 0;
  201:
  202:         //Count the number of edges connected to each partition.
  203:         count[ partvec[H.pins[i]] ]++;
  204:
  205:         //Nets which are connected only to cells in partition 0 belong in
partition 0.
  206:         //Nets which are connected only to cells in partition 1 belong in
partition 1.
  207:         //Nets which are connected to cells in partitions 0 and 1 were cut.
  208:         if( i+1==H.xpins[netcount+1] )
  209:         {
  210:             int total=0;
  211:             int which=0;
  212:             for( int j=0; j<K_PARTS && total<2; j++ ){
  213:                 if( count[j]!=0 ){
  214:                     which = j;
  215:                     total++;
  216:                 }
  217:             }
  218:             if( total==2 ) //Net is connected to at least two partitions.
  219:             {
  220:                 netparts[netcount] = K_PARTS; //------------Net was cut.
  221:             }
  222:             else
  223:             {
  224:                 netparts[netcount] = which; //--Net belons in partition
[which]
  225:                 netcounts[which] += 1; //-------Increment number of nets in
partition [which]
  226:                 edgecount[which] += count[which]; //--Increment number of
edges in partition [which]
  227:             }
  228:             netcount++; //Iterate to next net.
  229:         }
  230:     }
  231:     delete [] count;
  232:
  233:     // ********************************************************************
  234:     // After we know which cells and nets belong in each partition, we must
  235:     // create two separate hypergraphs to continue recursive partitioning.
  236:     // ********************************************************************
  237:
```

```
  238:     hypergraph G[ K_PARTS ];
  239:     int net[K_PARTS]; //----Counters for nets in each part
  240:     int pin[K_PARTS]; //----Counters for pins in each part
  241:     int vertx[K_PARTS]; //--Counters for vertices in each part
  242:
  243:     for( int i=0; i<K_PARTS; i++ ){
  244:         G[i].c = partweights[i];
  245:         G[i].n = netcounts[i];
  246:         G[i].xpins = new int [G[i].n+1];
  247:         G[i].pins  = new int [edgecount[i]];
  248:         G[i].nets  = new int [G[i].n];
  249:         G[i].vertx = new int [G[i].c];
  250:
  251:         net[i]=0;
  252:         pin[i]=0;
  253:         vertx[i]=0;
  254:     }
  255:     delete [] netcounts;
  256:     delete [] edgecount;
  257:     netcount = 0;
  258:
  259:     //PaToH requires the cells in each hypergraph to be numbered from 0 to
c-1.
  260:     //We store the original cell numbers in the vertx array.
  261:     int newCells[H.c];
  262:     for( int i=0; i<H.c; i++ )
  263:     {
  264:         G[ partvec[i] ].vertx[vertx[partvec[i]]] = H.vertx[i];
  265:         newCells[i] = vertx[partvec[i]]++;
  266:     }
  267:
  268:     //Now we store the net IDs, xpins, and pins for each partition.
  269:     for( int i=0; i<H.xpins[H.n]; i++ )
  270:     {
  271:         if( netparts[netcount]!=K_PARTS )
  272:         {
  273:             if( i==H.xpins[netcount] )
  274:             {
  275:                 G[ netparts[netcount] ].nets[net[netparts[netcount]]] =
H.nets[netcount];
  276:                 G[ netparts[netcount] ].xpins[net[netparts[netcount]]++] =
pin[netparts[netcount]];
  277:             }
  278:             G[ netparts[netcount] ].pins[pin[netparts[netcount]]++] =
newCells[H.pins[i]];
  279:         }
  280:         if( i+1==H.xpins[netcount+1] )
  281:             netcount++;
  282:     }
  283:
  284:     for( int i=0; i<K_PARTS; i++ )
  285:         G[i].xpins[ net[i] ] = pin[i];
  286:
  287:     delete [] partweights;
  288:     delete [] partvec;
  289:
  290:     //Continue recursive partitioning.
  291:     int sumNets=0, sumCells=0;
  292:     for( int i=0; i<K_PARTS; i++ ){
  293:         hund( G[i], P, Q,   constraints, K_PARTS,   partLevel+1, i,
nNets+sumNets, nCells+sumCells );
  294:         sumNets += G[i].n;
  295:         sumCells += G[i].c;
```

```
296:     }
297:
298:     for( int i=0; i<K_PARTS; i++ )
299:     {
300:         nNets += G[i].n;
301:
302:         if( nNets-1>=0 && nNets-1<Q.nc )
303:             constraints[ nNets-1 ] = 1;
304:     }
305:
306:     //Place the nets that were cut into matrix Q.
307:     for( int i=0; i<H.n; i++ )
308:     if( netparts[i]==K_PARTS )
309:         Q[nNets++] = H.nets[i];
310:
311:     if( nNets-1>=0 && nNets-1<Q.nc )
312:         constraints[ nNets-1 ] = 1;
313:
314:     delete[] netparts;
315: }
316:
317: #endif
318:
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    November 2011
    4:     File:    lu_super_dist.h
    5:     Purpose: This file contains function definitions for LU Factorization
with
    6:              SuperLU_DIST.
    7: */
    8:
    9: #ifndef LU_SUPER_DIST_H
   10: #define LU_SUPER_DIST_H
   11:
   12: #include "datastructures.h"
   13: #include "superlu_ddefs.h" //Makefile: -I$(DIR_SUPERLU_MT)/SRC
   14:
   15: class superLU_Dist_info
   16: {
   17:     public:
   18:         int nprocs; //Input
   19:         int nprow;  //Input
   20:         int npcol;  //Input
   21:         bool solve; //Input ; Solve or just factor
   22:         double *b;
   23:         double *x;
   24:         int status;
   25:         double acc1;
   26:         double acc2;
   27:         double acc3;
   28:
   29:
   30:         DiagScale_t DeFactoDiagScale; //Output
   31:         bool colReordered; //Output
   32:         bool rowReordered; //Output
   33:
   34:         superLU_Dist_info():
   35:             nprocs(0),
   36:             nprow(0),
   37:             npcol(0),
   38:             solve(false),
   39:             x(NULL),
   40:             b(NULL),
   41:             status(0),
   42:             acc1(0),
   43:             acc2(0),
   44:             acc3(0),
   45:             DeFactoDiagScale(NOEQUIL),
   46:             colReordered(false),
   47:             rowReordered(false){};
   48:         ~superLU_Dist_info()
   49:         {
   50:             if( x!=NULL ) { delete [] x; x = NULL; }
   51:             if( b!=NULL ) { delete [] b; b = NULL; }
   52:
   53:         };
   54:
   55:         void scale()
   56:         {
   57:             switch(DeFactoDiagScale)
   58:             {
   59:                 case NOEQUIL: printf("NOEQUIL"); break;
   60:                 case ROW:     printf("ROW");     break;
   61:                 case COL:     printf("COL");     break;
   62:                 case BOTH:    printf("BOTH");    break;
```

```
   63:                 default:      printf("Invalid"); break;
   64:             }
   65:         }
   66:
   67: };
   68:
   69: void compCol_to_compRowLoc( SuperMatrix *GA, SuperMatrix *A, int &ldb,
gridinfo_t *);
   70:
   71: // SuperLU_dist
   72: // 1. Initialize the SuperLU process grid
   73: // 2. Load the input matrix, convert to distributed format
   74: // 3. Configure the factorization options
   75: // 4. Factor
   76: void superLU_dist( sparseMatrix &A, superLU_Dist_info &SLUDinfo )
   77: {
   78:     SuperMatrix superGlobal_A,  //SType: SLU_NC,   column-wise, no supernode
   79:                 superLocal_A;   //SType: SLU_NR_loc, distributed compressed
row format
   80:
   81:     //Stype = SLU_NR_loc; Dtype = SLU_D; Mtype = SLU_GE.
   82:     //That is, A is stored in distributed compressed row format.
   83:
   84:     superlu_options_t  options;
   85:     SuperLUStat_t stat;
   86:     ScalePermstruct_t ScalePermstruct;
   87:     LUstruct_t LUstruct;
   88:     SOLVEstruct_t SOLVEstruct;
   89:     gridinfo_t grid;
   90:     int nrhs=0, m, n, info;
   91:     double *b=NULL;
   92:     double berr=0;
   93:     int ldb=0;
   94:     m = A.m;
   95:     n = A.n;
   96:
   97:     int m_loc, fst_row;
   98:
   99:
  100:
  101:     // 1. Initialize the SuperLU process grid
  102:     //
  103:     superlu_gridinit(MPI_COMM_WORLD, SLUDinfo.nprow, SLUDinfo.npcol, &grid);
  104:
  105:     // 2. Load the input matrix
  106:     //
  107:     dCreate_CompCol_Matrix_dist(&superGlobal_A, m, n, A.nnz, A.nzval,
A.adjncy, A.colptr, SLU_NC, SLU_D, SLU_GE);
  108:     compCol_to_compRowLoc(&superGlobal_A, &superLocal_A, ldb, &grid);
  109:     SUPERLU_FREE ( superGlobal_A.Store );
  110:
  111:
  112:     // Prepare the solution vectors
  113:     //
  114:     if( SLUDinfo.solve )
  115:     {
  116:         NRformat_loc *LAstore = (NRformat_loc*) superLocal_A.Store;
  117:         m_loc = LAstore->m_loc;
  118:         fst_row = LAstore->fst_row;
  119:
  120:
  121:         nrhs = 1;
  122:         ldb = m_loc;
```

```
  123:
  124:         //Generate Local B
  125:         //
  126:         b = new double[ m_loc ];
  127:         assert( b!=NULL );
  128:
  129:         for( int i=0; i<m_loc; i++ )
  130:             b[i] = SLUDinfo.b[fst_row + i];
  131:     }
  132:
  133:
  134:     // 3. Configure the factorization options
  135:     //
  136:     set_default_options_dist(&options);
  137:     /* Default input options:  full structure details in superlu_defs.h:490
  138:         set_default_options_dist(&options);
  139:
  140:         options.Fact              = DOFACT;             :{DOFACT,
SamePattern, SamePattern_SameRowPerm, FACTORED}
  141:         options.Equil             = YES;                :{NO, YES}
  142:         options.ParSymbFact       = NO;                 :{NO, YES}
  143:         options.ColPerm           = METIS_AT_PLUS_A;    :{NATURAL, MMD_ATA,
MMD_AT_PLUS_A, COLAMD,METIS_AT_PLUS_A, PARMETIS, ZOLTAN, MY_PERMC}
  144:         options.RowPerm           = LargeDiag;          :{NOROWPERM,
LargeDiag, MY_PERMR}
  145:         options.ReplaceTinyPivot  = YES;                :{NO, YES}
  146:         options.IterRefine        = DOUBLE;             :{NOREFINE,
SLU_SINGLE=1, SLU_DOUBLE, SLU_EXTRA}
  147:         options.Trans             = NOTRANS;            :{NOTRANS, TRANS,
CONJ}
  148:         options.SolveInitialized  = NO;                 :{NO, YES}
  149:         options.RefineInitialized = NO;                 :{NO, YES}
  150:         options.PrintStat         = YES;                :{NO, YES}
  151:         options->num_lookaheads    = 10;
  152:         options->lookahead_etree   = NO;
  153:         options->SymPattern        = NO;
  154:      */
  155:     options.Fact              = DOFACT;
  156:     options.Equil             = YES;
  157:     options.ParSymbFact       = NO;
  158:     options.ColPerm           = MMD_AT_PLUS_A; //NATURAL;
  159:     options.RowPerm           = LargeDiag;
  160:     options.ReplaceTinyPivot  = YES;
  161:     options.IterRefine        = SLU_DOUBLE;
  162:     options.Trans             = NOTRANS;
  163:     options.SolveInitialized  = NO;
  164:     options.RefineInitialized = NO;
  165:     options.PrintStat         = YES;
  166:     options.lookahead_etree   = NO;
  167:
  168:     // Initialize ScalePermstruct and LUstruct
  169:     //
  170:     ScalePermstructInit(m, n, &ScalePermstruct);
  171:     LUstructInit(m, n, &LUstruct);
  172:
  173:     // Initialize the statistics variables.
  174:     //
  175:     PStatInit(&stat);
  176:
  177:     // 4. Factor. [Call pdgssvx with nrhs=0]
  178:     //
  179:     pdgssvx(  &options, &superLocal_A, &ScalePermstruct, b, ldb, nrhs, &grid,
  180:               &LUstruct, &SOLVEstruct, &berr, &stat, &info );
```

```
  181:
  182:     SLUDinfo.status = info;
  183:
  184:     // Distribute Solution vector to Root Node
  185:     //
  186:     MPI_Status *tStatus=NULL;
  187:     if( grid.iam==0 )
  188:     {
  189:         // Copy solution to info.x
  190:         //
  191:         SLUDinfo.x = new double[ A.m ];
  192:         for( int i=0; i<m_loc; i++ )
  193:         {
  194:             SLUDinfo.x[i] = b[i];
  195:         }
  196:
  197:         int     start;
  198:         int     amount;
  199:         double *values;
  200:
  201:         // Receive solutions from other processes
  202:         for( int i=1; i<SLUDinfo.nprocs; i++ )
  203:         {
  204:             //Receive
  205:             //
  206:             MPI_Recv(&start,  1, MPI_INT,          i, 1, MPI_COMM_WORLD,
tStatus );
  207:             MPI_Recv(&amount, 1, MPI_INT,          i, 2, MPI_COMM_WORLD,
tStatus );
  208:
  209:             values = new double[ amount ];
  210:             assert(values!=NULL );
  211:
  212:             MPI_Recv( values, amount, MPI_DOUBLE,  i, 3, MPI_COMM_WORLD,
tStatus );
  213:
  214:             //Copy values to info.x
  215:             for( int j=0; j<amount; j++ )
  216:                 SLUDinfo.x[start+j] = values[j];
  217:
  218:             delete [] values;
  219:         }
  220:
  221:         // Compute Accuracy
  222:         //
  223:         int co1 = 0;
  224:         int co2 = 0;
  225:         int co3 = 0;
  226:         for( int i=0; i<A.m; i++ )
  227:         {
  228:             if( abs( SLUDinfo.x[i] - i ) < 1 )
  229:                 co1++;
  230:             if( abs( SLUDinfo.x[i] - i ) < 0.5 )
  231:                 co2++;
  232:             if( abs( SLUDinfo.x[i] - i ) < 0.2 )
  233:                 co3++;
  234:         }
  235:         SLUDinfo.acc1 = co1/(double)A.m*100;
  236:         SLUDinfo.acc2 = co2/(double)A.m*100;
  237:         SLUDinfo.acc3 = co3/(double)A.m*100;
  238:     }
  239:     else
  240:     {
```

```
  241:         //Transmit
  242:         //
  243:         MPI_Send(&fst_row,     1, MPI_INT,      0, 1, MPI_COMM_WORLD);
  244:         MPI_Send(  &m_loc,     1, MPI_INT,      0, 2, MPI_COMM_WORLD);
  245:         MPI_Send(       b, m_loc, MPI_DOUBLE,   0, 3, MPI_COMM_WORLD);
  246:     }
  247:
  248:
  249:     if( grid.iam==0 )
  250:     {
  251:         for( int i=0; i<m; i++ )
  252:             if( ScalePermstruct.perm_r[i]!=i )
  253:             {
  254:                 SLUDinfo.rowReordered = true;
  255:                 break;
  256:             }
  257:         for( int i=0; i<m; i++ )
  258:             if( ScalePermstruct.perm_c[i]!=i )
  259:             {
  260:                 SLUDinfo.colReordered = true;
  261:                 break;
  262:             }
  263:
  264:         //Check any irregularities here!
  265:         //See superlu_defs.h for ScalePermstruct_t definition.
  266:         //
  267:         SLUDinfo.DeFactoDiagScale = ScalePermstruct.DiagScale;
  268:
  269:         //
  270:         // Pc*Pr*diag(R)*A*diag(C)*Pc^T = L*U
  271:         // Pc: Column Permutation, reordering for sparcity and distribution
  272:         // Pr: Row Permutation, reordering for stability
  273:         // Diag(R): Diagonal of R, which is Scaling Vector?
  274:         // A: Input
  275:         // Diag(C): Diagonal of C, which is Scaling Vector?
  276:         // Pc^T:  Transpose of Column Permutation, for keeping diagonal
undisturbed.
  277:     }
  278:
  279:     //Clean up storage
  280:     //
  281:     PStatFree(&stat);
  282:     ScalePermstructFree(&ScalePermstruct);
  283:     Destroy_LU(n, &grid, &LUstruct);
  284:     LUstructFree(&LUstruct);
  285:     Destroy_CompRowLoc_Matrix_dist(&superLocal_A);
  286:
  287:     //Release the process grid
  288:     //
  289:     superlu_gridexit(&grid);
  290: }
  291:
  292:
  293: void compCol_to_compRowLoc( SuperMatrix *GA, SuperMatrix *A, int &ldb,
gridinfo_t *grid)
  294: {
  295:     //
  296:     //Code from SuperLU Example file: dcreate_matrix.c
  297:     //
  298:     int_t    *rowind, *colptr;   // global
  299:     double   *nzval;             // global
  300:     double   *nzval_loc;         // local
  301:     int_t    *colind, *rowptr;   // local
```

```
302:      int_t    m, n, nnz;
303:      int_t    m_loc, fst_row, nnz_loc;
304:      int_t    m_loc_fst;  // Record m_loc of the first p-1 processors,
305:                           // when mod(m, p) is not zero.
306:      int_t    iam, row, col, i, j, relpos;
307:      int_t    *marker;
308:
309:      NCformat *GAstore = (NCformat *) GA->Store;
310:      m      = GA->nrow;
311:      n      = GA->ncol;
312:      nnz    = GAstore->nnz;
313:      rowind = GAstore->rowind;
314:      colptr = GAstore->colptr;
315:      nzval  = (double*)GAstore->nzval;
316:
317:      iam = grid->iam;
318:
319:      /* Compute the number of rows to be distributed to local process */
320:      m_loc = m / (grid->nprow * grid->npcol);
321:      m_loc_fst = m_loc;
322:      /* When m / procs is not an integer */
323:      if ((m_loc * grid->nprow * grid->npcol) != m)
324:      {
325:          /*m_loc = m_loc+1;
326:          m_loc_fst = m_loc;*/
327:          if (iam == (grid->nprow * grid->npcol - 1)) /* last proc. gets all*/
328:              m_loc = m - m_loc * (grid->nprow * grid->npcol - 1);
329:      }
330:
331:
332:
333:      rowptr = (int_t *) intMalloc_dist(m_loc+1);
334:      marker = (int_t *) intCalloc_dist(n);
335:
336:      // Get counts of each row of GA
337:      //
338:      for (i = 0; i < n; ++i)
339:        for (j = colptr[i]; j < colptr[i+1]; ++j)
340:          ++marker[rowind[j]];
341:
342:      // Set up row pointers
343:      //
344:      rowptr[0] = 0;
345:      fst_row = iam * m_loc_fst;
346:      nnz_loc = 0;
347:      for (j = 0; j < m_loc; ++j)
348:      {
349:          row = fst_row + j;
350:          rowptr[j+1] = rowptr[j] + marker[row];
351:          marker[j] = rowptr[j];
352:      }
353:      nnz_loc = rowptr[m_loc];
354:
355:      nzval_loc = (double *) doubleMalloc_dist(nnz_loc);
356:      colind = (int_t *) intMalloc_dist(nnz_loc);
357:
358:      // Transfer the matrix into the compressed row storage
359:      //
360:      for (i = 0; i < n; ++i)
361:      {
362:          for (j = colptr[i]; j < colptr[i+1]; ++j)
363:          {
364:              row = rowind[j];
```

```
365:             if ( (row>=fst_row) && (row<fst_row+m_loc) )
366:             {
367:                 row = row - fst_row;
368:                 relpos = marker[row];
369:                 colind[relpos] = i;
370:                 nzval_loc[relpos] = nzval[j];
371:                 ++marker[row];
372:             }
373:         }
374:     }
375:
376:     ldb = m_loc;
377:
378:     // Set up the local A in NR_loc format
379:     //
380:     dCreate_CompRowLoc_Matrix_dist(A, m, n, nnz_loc, m_loc, fst_row,
381:                                    nzval_loc, colind, rowptr,
382:                                    SLU_NR_loc, SLU_D, SLU_GE);
383:     SUPERLU_FREE(marker);
384: }
385:
386: #endif
387:
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    October 2011
    4:     File:    main-bsld.cpp
    5:     Purpose: This file is used to perform LU decomposition.
    6:              By default, the program prints the number of non-zeros in L and
U
    7:              and the run-time for the LU factorization.
    8:     Usage:   bsld  [options]  block_size  input_matrix
    9:
   10:     Options:
   11:         -h                Print program usage [this message] and exit.
   12:         -p                Print the output matrices.
   13:         -d                Display the output matrices in an X11 window.
   14:         -s  [directory]   Save the output matrices to directory.
   15:
   16:     Arguments:
   17:         block_size        Maximum block size for hypergraph partitioning.
   18:         input_matrix      Filename of the matrix to be factored.
   19: */
   20:
   21: #include <cstdio>
   22: #include <string.h>
   23: #include <unistd.h>
   24: #include <ctype.h>
   25:
   26: #include "bsld.h"
   27: #include "timer.h"
   28: #include "out-x11.h"
   29: #include "verify.h"
   30:
   31: int main( int argc, char ** argv )
   32: {
   33: #ifdef _MPI
   34:     MPI::Init(argc,argv);
   35:     NUM_PROCS    = MPI::COMM_WORLD.Get_size();
   36:     NUM_PROCS_U  = (NUM_PROCS-1)/2;
   37:     NUM_PROCS_L  = NUM_PROCS - NUM_PROCS_U - 1;
   38:     PROCESS_RANK = MPI::COMM_WORLD.Get_rank();
   39:
   40:     if( NUM_PROCS <3 )
   41:     {
   42:         fprintf( stderr, "This program requires at least 3 processes!\n" );
   43:         MPI::Finalize();
   44:         return 2;
   45:     }
   46: #endif
   47:
   48:     int aflag = 0;
   49:     int pflag = 0;
   50:     int dflag = 0;
   51:     int sflag = 0;
   52:     char *svalue = NULL;
   53:     int c;
   54:
   55:     opterr = 0;
   56:     while ((c = getopt (argc, argv, "ahpds:")) != -1)
   57:     switch (c)
   58:     {
   59:         case 'h': printf("BSLD - Block Sparse LU Decomposition with
pivoting.\n\n" );
   60:                   printf("Usage: %s  [options]  block_size  input_matrix\n\n"
, basename(argv[0]) );
```

```
   61:                   printf("    Options:\n");
   62:                   printf("        -a                  Print the accuracy.\n");
   63:                   printf("        -h                  Print program usage
[this message] and exit.\n");
   64:                   printf("        -p                  Print the output
matrices.\n");
   65:                   printf("        -d                  Display the output
matrices in an X11 window.\n");
   66:                   printf("        -s  [directory]     Save the output
matrices to directory.\n\n");
   67:                   printf("    Arguments:\n");
   68:                   printf("        block_size          Maximum block size for
hypergraph partitioning.\n");
   69:                   printf("        input_matrix        Filename of the matrix
to be factored.\n");
   70:                   return 0;
   71:           case 'a': aflag = 1; break;
   72:           case 'p': pflag = 1; break;
   73:           case 'd': dflag = 1; break;
   74:           case 's': sflag = 1; svalue = optarg; break;
   75:           case '?':
   76:               if (optopt == 's')
   77:                   fprintf (stderr, "Option -%c requires an argument.\n",
optopt);
   78:               else if (isprint (optopt))
   79:                   fprintf (stderr, "Unknown option `-%c'.\n", optopt);
   80:               else
   81:                   fprintf (stderr, "Unknown option character `\\x%x'.\n",
optopt);
   82:               return 1;
   83:           default: abort();
   84:       }
   85:
   86:       solveStats stats;
   87:       int status = 0;
   88:       timer Time;
   89:
   90:       if( argc-optind!= 2 )
   91:       {
   92:           fprintf(stderr,"BSLD - Block Sparse LU Decomposition with
pivoting.\n\n" );
   93:           fprintf(stderr,"Usage: %s  [options]  block_size  input_matrix\n\n",
basename(argv[0]) );
   94:           fprintf(stderr,"    Options:\n");
   95:           fprintf(stderr,"        -a                  Print the accuracy.\n");
   96:           fprintf(stderr,"        -h                  Print program usage [this
message] and exit.\n");
   97:           fprintf(stderr,"        -p                  Print the output
matrices.\n");
   98:           fprintf(stderr,"        -d                  Display the output
matrices in an X11 window.\n");
   99:           fprintf(stderr,"        -s  [directory]     Save the output matrices
to directory.\n\n");
  100:           fprintf(stderr,"    Arguments:\n");
  101:           fprintf(stderr,"        block_size          Maximum block size for
hypergraph partitioning.\n");
  102:           fprintf(stderr,"        input_matrix        Filename of the matrix to
be factored.\n");
  103:
  104: #ifdef _MPI
  105:           MPI::Finalize();
  106: #endif
  107:
```

```
  108:         return 1;
  109:     }
  110:
  111:     int blockSize;
  112:     blockSize = atoi(argv[optind]);
  113:
  114: #ifdef _OPENMP
  115:     printf("%d Threads\n",omp_get_max_threads());
  116: #endif
  117:
  118:     sparseMatrix A, L, U;
  119:     if( A.rbRead( argv[optind+1] ) !=0 )
  120:     {
  121: #ifdef _MPI
  122:         MPI::Finalize();
  123: #endif
  124:
  125:         return 3;
  126:     }
  127:     rowPerm P(A.m);
  128:
  129:     TIME( Time, status = bsld( A, L, U, P, blockSize ) );
  130:
  131:     if( status!=0 )
  132:     {
  133: #ifdef _MPI
  134:         MPI::Finalize();
  135: #endif
  136:
  137:         return 0;
  138:     }
  139:
  140: #ifdef _MPI
  141:     if( PROCESS_RANK==0 )
  142:     {
  143: #endif
  144:         printf("L.nnz(): %d\n",L.nnz);
  145:         printf("U.nnz(): %d\n",U.nnz);
  146:         printf("%-22s %s\n","Pivot Replaced:",bsld_stats.replaced?"Yes":"No"
);
  147:         printf("Reordered: %c\n",BSLD_Reordered?'Y':'N');
  148:         printf("BSLD Time: %08.3lf\n", Time.getRunTime() );
  149:
  150:         sparseMatrix A2;
  151:         if( aflag|| pflag || dflag || sflag )
  152:             A2 = P*A;
  153:
  154:         if( aflag )
  155:         {
  156:             verify( A2, L, U, stats );
  157:             printf("Accuracy:  %.2lf%%\n", stats.acc);
  158:         }
  159:
  160:         if( pflag )
  161:         {
  162:             printf("A:  (%d)\n", A.nnz);  A.print();
  163:             printf("A2: (%d)\n",A2.nnz); A2.print();
  164:             printf("L:  (%d)\n", L.nnz);  L.print();
  165:             printf("U:  (%d)\n", U.nnz);  U.print();
  166:         }
  167:
  168:
  169:         if( dflag )
```

```
  170:         {
  171:             bool shade = true;
  172:             char winTitle[100];
  173:             sprintf( winTitle, "[ BSLD ][ %d ][ %s ]", blockSize,
basename(argv[optind+1]));
  174:             x11Window xwin( 200, winTitle );
  175:             xwin.addMatrix( A,  0, 0, shade, blockSize );
  176:             xwin.addMatrix( A2, 0, 1, shade, blockSize );
  177:             xwin.addMatrix( L,  1, 0, shade, blockSize );
  178:             xwin.addMatrix( U,  1, 1, shade, blockSize );
  179:             xwin.show();
  180:         }
  181:
  182:         if( sflag )
  183:         {
  184:             char filename[200];
  185:             char filename2[200];
  186:
  187:             sprintf( filename, "%s", basename(argv[optind+1]) );
  188:
  189:             sprintf( filename2,  "%s/%s-A2-%d.rb",svalue, filename,
blockSize );
  190:             A2.rbWrite( filename2 );
  191:
  192:             sprintf( filename2,  "%s/%s-L-%d.rb", svalue, filename,
blockSize );
  193:             L.rbWrite(filename2);
  194:
  195:             sprintf( filename2,  "%s/%s-U-%d.rb", svalue, filename,
blockSize );
  196:             U.rbWrite(filename2);
  197:         }
  198:
  199: #ifdef _MPI
  200:     }
  201:     MPI::Finalize();
  202: #endif
  203:     return 0;
  204: }
  205:
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    March 2012
    4:     File:    main-hund.cpp
    5:     Purpose: This file is used to reorder a matrix using the HUND algorithm.
    6:              By default, the program only prints the run-time for each step.
    7:     Usage:   hund  [options]  input_matrix  block_size  k_parts
    8:
    9:     Options:
   10:         -h               Print program usage [this message] and exit.
   11:         -d               Display the output matrices in an X11 window.
   12:         -s  [directory]  Save the output matrix to directory.
   13:
   14:     Arguments:
   15:         input_matrix     Filename of the matrix to be reordered.
   16:         block_size       Maximum block size for hypergraph partitioning.
   17:         k_parts          Number of parts in each recursive hypergraph
partition.
   18: */
   19:
   20: #include <cstdio>
   21: #include <string.h>
   22: #include <unistd.h>
   23: #include <ctype.h>
   24:
   25: #include "hund.h"
   26: #include "out-x11.h"
   27: #include "timer.h"
   28:
   29: int main( int argc, char ** argv)
   30: {
   31:     bool shade = true;
   32:     int dflag = 0;
   33:     int sflag = 0;
   34:     char *svalue = NULL;
   35:     int c;
   36:
   37:     opterr = 0;
   38:     while ((c = getopt (argc, argv, "hds:")) != -1)
   39:     switch (c)
   40:     {
   41:         case 'h': printf("HUND - Hypergraph Unsymmetric Nested Disection\n\n"
 );
   42:                   printf("Usage: %s  [options]  input_matrix  block_size
k_parts\n\n", basename(argv[0]) );
   43:                   printf("    Options:\n");
   44:                   printf("        -h                Print program usage
[this message] and exit.\n");
   45:                   printf("        -d                Display the output
matrices in an X11 window.\n");
   46:                   printf("        -s  [directory]   Save the output
matrices to directory.\n\n");
   47:                   printf("    Arguments:\n");
   48:                   printf("        input_matrix      Filename of the matrix
to be reordered.\n");
   49:                   printf("        block_size        Maximum block size for
hypergraph partitioning.\n");
   50:                   printf("        k_parts           Number of parts in each
recursive hypergraph partition.\n");
   51:                   return 0;
   52:         case 'd': dflag = 1; break;
   53:         case 's': sflag = 1; svalue = optarg; break;
   54:         case '?':
```

```
   55:             if (optopt == 's')
   56:                 fprintf (stderr, "Option -%c requires an argument.\n",
optopt);
   57:             else if (isprint (optopt))
   58:                 fprintf (stderr, "Unknown option `-%c'.\n", optopt);
   59:             else
   60:                 fprintf (stderr, "Unknown option character `\\x%x'.\n",
optopt);
   61:             return 1;
   62:         default: abort();
   63:     }
   64:
   65:     if( argc-optind!= 3 )
   66:     {
   67:         fprintf(stderr,"HUND - Hypergraph Unsymmetric Nested Disection\n\n"
);
   68:         fprintf(stderr,"Usage: %s  [options]  input_matrix  block_size
k_parts\n\n", basename(argv[0]) );
   69:         fprintf(stderr,"    Options:\n");
   70:         fprintf(stderr,"        -h                Print program usage [this
message] and exit.\n");
   71:         fprintf(stderr,"        -d                Display the output
matrices in an X11 window.\n");
   72:         fprintf(stderr,"        -s  [directory]   Save the output matrices
to directory.\n\n");
   73:         fprintf(stderr,"    Arguments:\n");
   74:         fprintf(stderr,"        input_matrix      Filename of the matrix to
be reordered.\n");
   75:         fprintf(stderr,"        block_size        Maximum block size for
hypergraph partitioning.\n");
   76:         fprintf(stderr,"        k_parts           Number of parts in each
recursive hypergraph partition.\n");
   77:         return 1;
   78:     }
   79:
   80:     MIN_BLOCK = atoi(argv[optind+1]);
   81:     int K_PARTS   = atoi(argv[optind+2]);
   82:
   83:     timer Time;
   84:
   85:     sparseMatrix A;
   86:     if( A.rbRead( argv[optind] ) !=0 )
   87:         return 2;
   88:
   89:     rowPerm P(A.m);
   90:     colPerm Q(A.n);
   91:
   92:     //Convert A to hypergraph
   93:     //
   94:     hypergraph *H = new hypergraph;
   95:     assert(H!=NULL);
   96:     *H = A;
   97:
   98:
   99:     //Initialize COLAMD constraints
  100:     //
  101:     int* colamd_constraints = new int[A.n];
  102:     assert(colamd_constraints!=NULL);
  103:     for( int i=0; i<A.n; i++ )
  104:         colamd_constraints[i]=0;
  105:
  106:
  107:     printf("Size: %d x %d\n", A.m, A.n);
```

```
  108:      printf(" NNZ: %d\n", A.nnz);
  109:      printf("   B: %d\n", MIN_BLOCK);
  110:      printf("   K: %d\n", K_PARTS  );
  111:
  112:
  113:      char winTitle[100];
  114:      sprintf( winTitle, "[ HUND-SuLU ][ B%d ][ K%d ][ %s ]", MIN_BLOCK,
K_PARTS, basename(argv[optind]) );
  115:      x11Window xwin( 200, winTitle );
  116:      if( dflag )
  117:          xwin.addMatrix( A, 0, 0, shade, MIN_BLOCK );
  118:
  119:
  120:      //Generate P and Q
  121:      //
  122:      printf("%-22s ","HUND...");
  123:      TIME( Time, hund(*H, P, Q, colamd_constraints, K_PARTS ) );
  124:      printf("%08.3lf\n", Time.getRunTime());
  125:
  126:      printf("%-22s ","HUND Reordering...");
  127:      TIME( Time, A = P*A*Q );
  128:      printf("%08.3lf\n", Time.getRunTime());
  129:
  130:
  131:      if( sflag )
  132:      {
  133:          char filebase[128];
  134:          char filename[128];
  135:          sprintf( filebase, "%s", basename( argv[optind] ) );
  136:
  137:          sprintf(filename, "%s/%s-HUND_B%02d_K%02d.rb"
,svalue,filebase,MIN_BLOCK,K_PARTS);
  138:          A.rbWrite(filename);
  139:      }
  140:
  141:
  142:      if( dflag )
  143:          xwin.addMatrix( A, 1, 0, shade, MIN_BLOCK );
  144:
  145:      //Generate P2
  146:      //
  147:      printf("%-22s ","MC64... ");
  148:      TIME( Time, mc64( A, P ) );
  149:      printf("%08.3lf\n", Time.getRunTime());
  150:
  151:      printf("%-22s ","MC64 Reordering... ");
  152:      TIME( Time, A = P*A );
  153:      printf("%08.3lf\n", Time.getRunTime());
  154:
  155:      if( dflag )
  156:          xwin.addMatrix( A, 1, 1, shade, MIN_BLOCK );
  157:
  158:
  159:      if( sflag )
  160:      {
  161:          char filebase[128];
  162:          char filename[128];
  163:          sprintf( filebase, "%s", basename( argv[optind] ) );
  164:
  165:          sprintf(filename, "%s/%s-MC64_B%02d_K%02d.rb"
,svalue,filebase,MIN_BLOCK,K_PARTS);
  166:          A.rbWrite(filename);
  167:      }
```

```
  168:
  169:
  170:      //Generate P3 and P3T
  171:      //
  172:      printf("%-22s ","CCOLAMD... ");
  173:      TIME( Time, cColamd( A, P, Q, colamd_constraints ) );
  174:      printf("%08.3lf\n", Time.getRunTime());
  175:
  176:      printf("%-22s ","CCOLAMD Reordering... ");
  177:      TIME( Time, A = P*A*Q );
  178:      printf("%08.3lf\n", Time.getRunTime());
  179:      delete[] colamd_constraints;
  180:
  181:
  182:      printf("%-22s %08.3lf\n","Total Time",Time.getTotalTime());
  183:
  184:
  185:      if( dflag )
  186:      {
  187:          xwin.addMatrix( A, 1, 2, shade, MIN_BLOCK );
  188:
  189:          int pid = fork();
  190:          if( pid==0 )
  191:          {
  192:              xwin.show();
  193:              return 0;
  194:          }
  195:      }
  196:
  197:      if( sflag )
  198:      {
  199:          char filebase[128];
  200:          char filename[128];
  201:          sprintf( filebase, "%s", basename( argv[optind] ) );
  202:
  203:          sprintf(filename, "%s/%s-CCOLAMD_B%02d_K%02d.rb"
,svalue,filebase,MIN_BLOCK,K_PARTS);
  204:          A.rbWrite(filename);
  205:      }
  206:
  207:      return 0;
  208: }
  209:
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    November 2011
    4:     File:    main-lu-super-dist.cpp
    5:     Purpose: This file is used to perform LU decomposition
    6:              for verification purposes.
    7:     Usage:   lu-super-dist  input_matrix
    8: */
    9:
   10: #include <cstdio>
   11: #include <mpi.h>
   12: #include <unistd.h>
   13:
   14: #include "lu_super_dist.h"
   15: #include "timer.h"
   16:
   17: int main( int argc, char ** argv )
   18: {
   19:     MPI_Init(&argc,&argv);
   20:     int nproc;
   21:     int rank;
   22:     MPI_Comm_size(MPI_COMM_WORLD,&nproc);
   23:     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   24:
   25:     int aflag = 0;
   26:     int c;
   27:
   28:     opterr = 0;
   29:     while ((c = getopt (argc, argv, "ah:")) != -1)
   30:     switch (c)
   31:     {
   32:         case 'h': if(rank==0)
   33:                   {
   34:                       printf("LU-Super-DIST - Super LU Decomposition.
[Distributed Version]\n\n" );
   35:                       printf("Usage: %s  [options]  input_matrix\n\n",
basename(argv[0]) );
   36:                       printf("    Options:\n");
   37:                       printf("        -a                Print the
accuracy.\n");
   38:                       printf("        -h                Print program usage
[this message] and exit.\n");
   39:                       printf("    Arguments:\n");
   40:                       printf("        input_matrix      Filename of the
matrix to be factored.\n");
   41:                   }
   42:                   MPI_Finalize();
   43:                   return 0;
   44:         case 'a': aflag = 1; break;
   45:         case '?':
   46:             if (optopt == 's')
   47:                 fprintf (stderr, "Option -%c requires an argument.\n",
optopt);
   48:             else if (isprint (optopt))
   49:                 fprintf (stderr, "Unknown option `-%c'.\n", optopt);
   50:             else
   51:                 fprintf (stderr, "Unknown option character `\\x%x'.\n",
optopt);
   52:             MPI_Finalize();
   53:             return 1;
   54:         default: abort();
   55:     }
   56:
```

```
    57:     if( argc-optind!= 1 )
    58:     {
    59:         if( rank==0 )
    60:         {
    61:             fprintf(stderr,"LU-Super-DIST - Super LU Decomposition.
[Distributed Version]\n\n" );
    62:             fprintf(stderr,"Usage: %s  [options]  input_matrix\n\n",
basename(argv[0]) );
    63:             fprintf(stderr,"    Options:\n");
    64:             fprintf(stderr,"        -a                  Print the accuracy.\n"
);
    65:             fprintf(stderr,"        -h                  Print program usage
[this message] and exit.\n\n");
    66:             fprintf(stderr,"    Arguments:\n");
    67:             fprintf(stderr,"        input_matrix        Filename of the
matrix to be factored.\n");
    68:         }
    69:
    70:         MPI_Finalize();
    71:         return 1;
    72:     }
    73:
    74:
    75:
    76:     timer LUTimer;
    77:
    78:
    79:     sparseMatrix A, L, U;
    80:
    81:
    82:     if( A.rbRead( argv[optind] ) !=0 )
    83:     {
    84:         MPI_Finalize();
    85:         return 2;
    86:     }
    87:
    88:
    89:
    90:     superLU_Dist_info info;
    91:     info.nprocs = nproc;
    92:     info.nprow  = nproc;
    93:     info.npcol  = 1;
    94:
    95:     //info.nprow  = 2;
    96:     //info.npcol  = nproc/2;
    97:
    98:     if( aflag==1 )
    99:     {
   100:         info.solve = true;
   101:
   102:         sparseMatrix *X;
   103:         denseMatrix B;
   104:
   105:         //Create X vector
   106:         //
   107:         X = new sparseMatrix(A.m,1,A.m);
   108:         assert(X!=NULL);
   109:         X->colptr[0] = 0;
   110:         for( int i=0; i<A.m; i++ )
   111:         {
   112:             X->nzval[i]  = i;
   113:             X->adjncy[i] = i;
   114:         }
```

```
115:
116:          //Multiply A*X = B
117:          //
118:          B = A*(*X);
119:          delete X;
120:
121:          //Copy B vector from sparseMatrix format to vector
122:          //
123:          info.b = new double[ A.m ];
124:          assert( info.b!=NULL );
125:          for( int i=0; i<A.m; i++ )
126:              info.b[i] = B[i][0];
127:      }
128:
129:
130:      if( rank==0 )
131:          LUTimer.start();
132:
133:      superLU_dist( A, info );
134:
135:      if( rank==0 )
136:      {
137:          LUTimer.stop();
138:
139:          printf("    nproc: %d\n", nproc);
140:
141:          printf("    C: %c\n", info.colReordered?'Y':'N');
142:          printf("    R: %c\n", info.rowReordered?'Y':'N');
143:          printf("    DiagScale: "); info.scale(); printf("\n");
144:
145:          if( info.solve )
146:          {
147:              printf("    Accuracy1: %.2lf%%\n",info.acc1);
148:              printf("    Accuracy2: %.2lf%%\n",info.acc2);
149:              printf("    Accuracy3: %.2lf%%\n",info.acc3);
150:              printf("    Status: %d\n",info.status);
151:          }
152:
153:          printf("    LU Time: %08.3lf\n", LUTimer.getRunTime());
154:      }
155:
156:      MPI_Finalize();
157:      return 0;
158: }
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    July 2010
    4:     File:    main-util-rb2png.cpp
    5:     Purpose: This file is used to convert a matrix in Rutherford-Boeing
format
    6:              to a PNG image.
    7:     Usage:   rb2png  [options]  input_file  output_file
    8:
    9:     Options:
   10:         -h              Print program usage and exit
   11:         -s              Use pixel shading
   12:         -w   [width]    Image width
   13: */
   14:
   15: #include <unistd.h>
   16: #include <cstdio>
   17: #include <cstdlib>
   18: #include "datastructures.h"
   19: #include "out-png.h"
   20:
   21: int main( int argc, char ** argv)
   22: {
   23:     int  width  = 0;
   24:     int  height = 0;
   25:     bool sflag  = false;
   26:     int c;
   27:
   28:     opterr = 0;
   29:     while ((c = getopt (argc, argv, "hsw:")) != -1)
   30:     switch (c)
   31:     {
   32:         case 'h': printf("rb2png - RB matrix file to PNG image
converter.\n\n" );
   33:                   printf("Usage: %s  [options]  input_file  output_file\n\n"
, basename(argv[0]) );
   34:                   printf("    Options:\n");
   35:                   printf("        -h                Print program usage
[this message] and exit.\n");
   36:                   printf("        -s                Use pixel shading.\n");
   37:                   printf("        -w  [width]       Image width.\n");
   38:                   printf("    Arguments:\n");
   39:                   printf("        input_file        Input file in
Rutherford-Boeing format.\n");
   40:                   printf("        output_file       Output filename.\n");
   41:                   return 0;
   42:         case 'w': width = atoi(optarg); break;
   43:         case 's': sflag = true; break;
   44:         case '?':
   45:             if (optopt == 'w')
   46:                 fprintf (stderr, "Option -%c requires an argument.\n",
optopt);
   47:             else if (isprint (optopt))
   48:                 fprintf (stderr, "Unknown option `-%c'.\n", optopt);
   49:             else
   50:                 fprintf (stderr, "Unknown option character `\\x%x'.\n",
optopt);
   51:             return 1;
   52:         default: abort();
   53:     }
   54:
   55:     if( argc-optind!= 2 )
   56:     {
```

```
   57:         fprintf(stderr,"rb2png - RB matrix file to PNG image converter.\n\n"
);
   58:         fprintf(stderr,"Usage: %s  [options]  input_file  output_file\n\n",
basename(argv[0]) );
   59:         fprintf(stderr,"    Options:\n");
   60:         fprintf(stderr,"        -h                  Print program usage [this
message] and exit.\n");
   61:         fprintf(stderr,"        -s                  Use pixel shading.\n");
   62:         fprintf(stderr,"        -w  [width]         Image width.\n");
   63:         fprintf(stderr,"    Arguments:\n");
   64:         fprintf(stderr,"        input_file          Input file in
Rutherford-Boeing format.\n");
   65:         fprintf(stderr,"        output_file         Output filename.\n");
   66:         return 1;
   67:     }
   68:
   69:
   70:     if( width>0 && height<=0 )
   71:         height = width;
   72:     if( height>0 && width<=0 )
   73:         width = height;
   74:     if( width<=0 && height<=0 )
   75:         width = height = 200;
   76:
   77:     sparseMatrix A;
   78:     if( A.rbRead( argv[optind] ) !=0 )
   79:         return 3;
   80:
   81:     int code = writeImage( argv[optind+1], width, height, A, sflag );
   82:
   83:     if( code!=0 ){
   84:         fprintf( stderr, "out-png error: %d!\n", code );
   85:         return code;
   86:     }
   87:
   88:     return 0;
   89: }
   90:
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    June 2013
    4:     File:    main-util-rbconv.cpp
    5:     Purpose: This file is used to convert Rutherford-Boeing matrix files to
human-readable format.
    6:     Usage:   rbconv  [options]  input  output
    7:
    8:     Options:
    9:         -h              Print program usage and exit.
   10: */
   11:
   12: #include <cstdio>
   13: #include <unistd.h>
   14:
   15: #include "datastructures.h"
   16:
   17: int main( int argc, char ** argv)
   18: {
   19:     int c;
   20:
   21:     opterr = 0;
   22:     while ((c = getopt (argc, argv, "h")) != -1)
   23:     switch (c)
   24:     {
   25:         case 'h': printf("rbconv - RB matrix converter.\n\n" );
   26:                   printf("Usage: %s  [options]  input  output\n\n",
basename(argv[0]) );
   27:                   printf("    Options:\n");
   28:                   printf("        -h                  Print program usage
[this message] and exit.\n");
   29:                   printf("    Arguments:\n");
   30:                   printf("        input     Input filename.\n");
   31:                   printf("        output    Output filename.\n");
   32:                   return 0;
   33:         case '?':
   34:             if (isprint (optopt))
   35:                 fprintf (stderr, "Unknown option `-%c'.\n", optopt);
   36:             else
   37:                 fprintf (stderr, "Unknown option character `\\x%x'.\n",
optopt);
   38:             return 1;
   39:         default: abort();
   40:     }
   41:
   42:     if( argc-optind!= 2 )
   43:     {
   44:         fprintf(stderr,"rbconv - RB matrix converter.\n\n" );
   45:         fprintf(stderr,"Usage: %s  [options]  input  output\n\n",
basename(argv[0]) );
   46:         fprintf(stderr,"    Options:\n");
   47:         fprintf(stderr,"        -h                  Print program usage [this
message] and exit.\n");
   48:         fprintf(stderr,"    Arguments:\n");
   49:         fprintf(stderr,"        input     Input filename.\n");
   50:         fprintf(stderr,"        output    Output filename.\n");
   51:         return 1;
   52:     }
   53:
   54:
   55:     // 1. Determine whether input is in RB format or human-readable
   56:     // 2. Determine whether output will be RB format or human-readable
   57:     // 3. Perform appropriate conversion
```

```
    58:
    59:     sparseMatrix S;
    60:     denseMatrix D;
    61:     int err=0;
    62:     bool arg1rb = false;
    63:     bool arg2rb = false;
    64:     int arg1len = strlen(  argv[optind]   );
    65:     int arg2len = strlen(  argv[optind+1] );
    66:
    67:     // 1. Determine whether input is in RB format or human-readable
    68:     //
    69:     if( strcmp( argv[optind]+arg1len-3, ".rb" )==0 )
    70:         arg1rb = true;
    71:
    72:     // 2. Determine whether output will be RB format or human-readable
    73:     //
    74:     if( strcmp( argv[optind+1]+arg2len-3, ".rb" )==0 )
    75:         arg2rb = true;
    76:
    77:     // Error handling
    78:     //
    79:     if(  arg1rb && arg2rb || !arg1rb && !arg2rb )
    80:     {
    81:         fprintf( stderr, "Conversion must be [.rb to .txt]  or  [.txt to
.rb]\n" );
    82:         return 1;
    83:     }
    84:     // 3. Perform appropriate conversion
    85:     //
    86:     if(  arg1rb )
    87:     {
    88:         // .rb to .txt conversion
    89:         //
    90:         if( S.rbRead( argv[optind] ) )
    91:         {
    92:             fprintf( stderr, "Could not open input file: %s\n", argv[optind]
);
    93:             return 1;
    94:         }
    95:
    96:         D = S;
    97:         D.xport( argv[optind+1] );
    98:     }
    99:     else
   100:     {
   101:         // .txt to .rb conversion
   102:         //
   103:         err = D.import( argv[optind] );
   104:         if( err )
   105:         {
   106:             fprintf( stderr, "Could not open input file: %s\n", argv[optind]
);
   107:             fprintf( stderr, "Error %d\n", err );
   108:             return 1;
   109:         }
   110:
   111:         S = D;
   112:         S.rbWrite( argv[optind+1] );
   113:     }
   114:
   115:     return 0;
   116: }
   117:
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    September 2011
    4:     File:    main-util-rbls.cpp
    5:     Purpose: This file is used to get the size and nnz of Rutherford-Boeing
matrices.
    6:     Inputs:  ARG1=<Matrix Filename>, ARG2=<Matrix Filename> ...
    7:     Output:  STDOUT= m x n   nnz   filename...
    8: */
    9:
   10: #include <cstdio>
   11: #include <cstdlib>
   12: #include "datastructures.h"
   13:
   14: int main( int argc, char ** argv)
   15: {
   16:     for( int i=1; i<argc; i++ )
   17:     {
   18:         sparseMatrix A;
   19:         if( A.rbRead( argv[i] ) !=0 )
   20:             continue;
   21:
   22:         printf( " %8d x %-8d %9d  %9d  %s\n", A.m, A.n, A.nnz, A.dz(),
argv[i] );
   23:     }
   24:
   25:     return 0;
   26: }
   27:
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    August 2010
    4:     File:    main-util-rbprint.cpp
    5:     Purpose: This file is used to print out a Rutherford-Boeing matrix in
dense
    6:              form.
    7:     Usage:   rbprint  [options]  filename
    8:
    9:     Options:
   10:         -h              Print program usage [this message] and exit.
   11:         -d              Print values in decimal format.
   12:         -e              Print values in exponential format [default]
   13:         -f              Print values in floating point format.
   14:     Inputs:  ARG1=<Matrix Filename>
   15:     Output:  STDOUT = Size: Rows x Cols, NNZ: nnz
   16:                       Dense matrix print-out.
   17: */
   18:
   19: #include <cstdio>
   20: #include <unistd.h>
   21:
   22: #include "datastructures.h"
   23:
   24: int main( int argc, char ** argv)
   25: {
   26:     int c;
   27:     int fmt = 2;
   28:
   29:     opterr = 0;
   30:     while ((c = getopt (argc, argv, "hdef")) != -1)
   31:     switch (c)
   32:     {
   33:         case 'h': printf("rbprint - RB matrix printer.\n\n" );
   34:                   printf("Usage: %s  [options]  filename\n\n",
basename(argv[0]) );
   35:                   printf("    Options:\n");
   36:                   printf("        -h                Print program usage
[this message] and exit.\n");
   37:                   printf("        -d                Print values in decimal
format.\n");
   38:                   printf("        -e                Print values in
exponential format [default]\n");
   39:                   printf("        -f                Print values in
floating point format\n");
   40:                   printf("    Arguments:\n");
   41:                   printf("        filename          Input filename.\n");
   42:                   return 0;
   43:         case 'd': fmt=1; break;
   44:         case 'e': fmt=2; break;
   45:         case 'f': fmt=3; break;
   46:         case '?':
   47:             if (isprint (optopt))
   48:                 fprintf (stderr, "Unknown option `-%c'.\n", optopt);
   49:             else
   50:                 fprintf (stderr, "Unknown option character `\\x%x'.\n",
optopt);
   51:             return 1;
   52:         default: abort();
   53:     }
   54:
   55:     if( argc-optind!= 1 )
   56:     {
```

```
   57:         fprintf(stderr,"rbprint - RB matrix printer.\n\n" );
   58:         fprintf(stderr,"Usage: %s  [options]  filename\n\n",
basename(argv[0]) );
   59:         fprintf(stderr,"    Options:\n");
   60:         fprintf(stderr,"        -h                  Print program usage [this
message] and exit.\n");
   61:         fprintf(stderr,"        -d                  Print values in decimal
format.\n");
   62:         fprintf(stderr,"        -e                  Print values in
exponential format [default]\n");
   63:         fprintf(stderr,"        -f                  Print values in floating
point format\n");
   64:         fprintf(stderr,"    Arguments:\n");
   65:         fprintf(stderr,"        filename            Input filename.\n");
   66:         return 1;
   67:     }
   68:
   69:     sparseMatrix A;
   70:
   71:     if( A.rbRead( argv[optind] ) !=0 )
   72:         return 3;
   73:
   74:     printf( "Size: %dx%d,  NNZ: %d\n", A.m, A.n, A.nnz );
   75:
   76:
   77:     switch( fmt )
   78:     {
   79:         case 1:  A.print((char*)"%3.0f ", (char*)"%3d "); break;
   80:         case 2:  A.print((char*)"%9.2e ", (char*)"%9d "); break;
   81:         case 3:  A.print((char*)"%8.4f  ",(char*)"%8d  "); break;
   82:         default: A.print((char*)"%8.1e ", (char*)"%8d "); break;
   83:     }
   84:
   85:     return 0;
   86: }
   87:
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    September 2011
    4:     File:    main-util-rbview.cpp
    5:     Purpose: This file is used to view Rutherford-Boeing matrices.
    6:     Usage:   rbview  [options]  input_file...
    7:
    8:     Options:
    9:         -h              Print program usage and exit.
   10:         -s              Use pixel shading
   11:         -w  [width]     Image width
   12: */
   13:
   14: #include <cstdio>
   15: #include <cstdlib>
   16: #include <unistd.h>
   17: #include <string.h>
   18:
   19: #include "datastructures.h"
   20: #include "out-x11.h"
   21:
   22: int main( int argc, char ** argv)
   23: {
   24:     int  width  = 0;
   25:     bool sflag  = false;
   26:     int c;
   27:
   28:     opterr = 0;
   29:     while ((c = getopt (argc, argv, "hsw:")) != -1)
   30:     switch (c)
   31:     {
   32:         case 'h': printf("rbview - X11 RB matrix file viewer.\n\n" );
   33:                   printf("Usage: %s  [options]  input_file...\n\n",
basename(argv[0]) );
   34:                   printf("    Options:\n");
   35:                   printf("        -h                  Print program usage
[this message] and exit.\n");
   36:                   printf("        -s                  Use pixel shading.\n");
   37:                   printf("        -w  [width]         Image width.\n");
   38:                   printf("    Arguments:\n");
   39:                   printf("        input_file          Input file in
Rutherford-Boeing format.\n");
   40:                   return 0;
   41:         case 'w': width = atoi(optarg); break;
   42:         case 's': sflag = true; break;
   43:         case '?':
   44:             if (optopt == 'w')
   45:                 fprintf (stderr, "Option -%c requires an argument.\n",
optopt);
   46:             else if (isprint (optopt))
   47:                 fprintf (stderr, "Unknown option `-%c'.\n", optopt);
   48:             else
   49:                 fprintf (stderr, "Unknown option character `\\x%x'.\n",
optopt);
   50:             return 1;
   51:         default: abort();
   52:     }
   53:
   54:     if( width<=0 )
   55:         width = 200;
   56:
   57:
   58:     vector<string> files;
```

```
    59:     for( int i=optind; i<argc; i++ )
    60:         files.push_back( string(argv[i]) );
    61:
    62:     int pid = fork();
    63:     if( pid==0 )
    64:     {
    65:         x11Window xwin( width, files, sflag );
    66:         xwin.show();
    67:     }
    68:
    69:     return 0;
    70: }
    71:
```

```
    1: DIR_BINARIES     =$(HOME)/testbench/bin
    2: DIR_LIB          =$(HOME)/sparse
    3: DIR_PATOH        =$(DIR_LIB)/PaToH
    4: DIR_SUPERLU      =$(DIR_LIB)/SuperLU_4.2
    5: DIR_SUPERLU_MT   =$(DIR_LIB)/SuperLU_MT_2.0
    6: DIR_SUPERLU_DIST =$(DIR_LIB)/SuperLU_DIST_3.0
    7: DIR_SUITESPARSE  =$(DIR_LIB)/SuiteSparse
    8: DIR_METIS        =$(DIR_LIB)/parmetis-4.0.2
    9: DIR_GOTOBLAS     =$(HOME)/lib/GotoBLAS2
   10: DIR_CBLAS        =$(HOME)/lib/CBLAS
   11:
   12: #Header Files
   13: H_PATOH        = -I$(DIR_PATOH)
   14: H_SUPERLU      = -I$(DIR_SUPERLU)/SRC
   15: H_SUPERLU_MT   = -I$(DIR_SUPERLU_MT)/SRC
   16: H_SUPERLU_DIST = -I$(DIR_SUPERLU_DIST)/SRC
   17: H_UFCONF       = -I$(DIR_SUITESPARSE)/UFconfig
   18: H_RBIO         = -I$(DIR_SUITESPARSE)/RBio/Include
   19: H_CCOLAMD      = -I$(DIR_SUITESPARSE)/CCOLAMD/Include
   20: H_METIS        = -I$(DIR_METIS)/include
   21: HEADERS_RB           = $(H_UFCONF) $(H_RBIO)
   22: HEADERS_HUND         = $(HEADERS_RB) $(H_CCOLAMD) $(H_PATOH) $(H_SUPERLU)
   23: HEADERS_SUPERLU      = $(HEADERS_RB) $(H_SUPERLU)
   24: HEADERS_SUPERLU_MT   = $(HEADERS_RB) $(H_SUPERLU_MT)
   25: HEADERS_SUPERLU_DIST = $(HEADERS_RB) $(H_SUPERLU_DIST)
   26:
   27: #Libraries
   28: C_BLAS           = -I$(DIR_CBLAS)/include  $(DIR_CBLAS)/lib/cblas_LINUX.a
   29: GSL_BLAS         = -L/usr/lib/blas/reference -lblas -lcblas
   30: GOTO_BLAS        = -I$(DIR_GOTOBLAS) $(DIR_GOTOBLAS)/libgoto2.a -lgfortran
-lpthread
   31: LIB_BLAS         = $(GSL_BLAS)
   32:
   33: LIB_PATOH        = $(DIR_PATOH)/libpatoh.a -lm
   34: LIB_SUPERLU      = -L$(DIR_SUPERLU)/lib -lsuperlu_4.2 -lgfortran
   35: LIB_SUPERLU_MT   = -L$(DIR_SUPERLU_MT)/lib -lsuperlu_mt_PTHREAD -lgfortran
-lpthread
   36: LIB_SUPERLU_DIST = $(DIR_SUPERLU_DIST)/lib/libsuperlu_dist_3.0.a
   37: LIB_UFCONF       = $(DIR_SUITESPARSE)/UFconfig/libufconfig.a
   38: LIB_RBIO         = $(DIR_SUITESPARSE)/RBio/Lib/librbio.a
   39: LIB_CCOLAMD      = $(DIR_SUITESPARSE)/CCOLAMD/Lib/libccolamd.a
   40: LIB_METIS        = $(DIR_METIS)/build/Linux-x86_64/libparmetis/libparmetis.a
\
   41:                    $(DIR_METIS)/build/Linux-x86_64/libmetis/libmetis.a
   42: LIBS_RB           = $(LIB_BLAS) $(LIB_UFCONF) $(LIB_RBIO)
   43: LIBS_HUND         = $(LIBS_RB) $(LIB_CCOLAMD) $(LIB_PATOH) $(LIB_SUPERLU)
   44: LIBS_SUPERLU      = $(LIBS_RB) $(LIB_SUPERLU)
   45: LIBS_SUPERLU_MT   = $(LIBS_RB) $(LIB_SUPERLU_MT)
   46: LIBS_SUPERLU_DIST = $(LIBS_RB) $(LIB_SUPERLU_DIST) $(LIB_METIS) $(LIB_BLAS)
   47:
   48: CPP = g++
   49: MPI_CPP = mpic++
   50:
   51: OPTIONS = -Wextra -D_BSLD_ZERO_CHECK -g
   52:
   53: all: superlu bsld utility
   54:
   55: superlu: lu-super-dist
   56:
   57: bsld: bsld-lin-partial \
   58:       bsld-lin-threshold \
   59:       bsld-omp-partial \
   60:       bsld-omp-threshold \
```

```
   61:       bsld-mpi-partial \
   62:       bsld-mpi-threshold
   63:
   64: utility: hund \
   65:           rb2png \
   66:           rbprint \
   67:           rbls \
   68:           rbview \
   69:           rbconv
   70:
   71:
   72: clean:
   73:         @echo \# Clean
   74:         rm -v $(DIR_BINARIES)/*
   75:
   76:
   77: # Object files ---------------------------------------------------------
   78: #
   79: OBJ_DATASTRUCTURES = $(DIR_BINARIES)/datastructures.o
   80: OBJ_BLOCKMATRIX    = $(DIR_BINARIES)/blockmatrix.o
   81: OBJ_X11            = $(DIR_BINARIES)/out-x11.o
   82: OBJ_BSLD           = $(OBJ_DATASTRUCTURES) $(OBJ_BLOCKMATRIX) $(OBJ_X11)
   83:
   84: $(OBJ_DATASTRUCTURES):  datastructures.cpp datastructures.h
   85:         @echo \# 'basename $@'
   86:         $(CPP) -c $(OPTIONS) -o $@ $<  $(HEADERS_RB)
   87:         @echo
   88:
   89: $(OBJ_BLOCKMATRIX):  blockmatrix.cpp blockmatrix.h $(OBJ_DATASTRUCTURES)
   90:         @echo \# 'basename $@'
   91:         $(CPP) -c $(OPTIONS) -o $@ $<  $(HEADERS_RB)
   92:         @echo
   93:
   94: $(OBJ_X11):  out-x11.cpp out-x11.h $(OBJ_DATASTRUCTURES)
   95:         @echo \# 'basename $@'
   96:         $(CPP) -c $(OPTIONS) -o $@ $<  $(HEADERS_RB)
   97:         @echo
   98:
   99:
  100:
  101: # SuperLU Program -------------------------------------------------------
  102: #
  103: lu-super-dist:  $(DIR_BINARIES)/lu-super-dist
  104:
  105: $(DIR_BINARIES)/lu-super-dist: main-lu-super-dist.cpp lu_super_dist.h $(
OBJ_DATASTRUCTURES) $(OBJ_X11)
  106:         @echo \# 'basename $@'
  107:         @echo . /opt/gcdistro/modules/Modules/init/bash
  108:         @echo module purge
  109:         @echo module load mpi/openmpi
  110:         $(MPI_CPP) $(OPTIONS) -o $@ $< $(OBJ_DATASTRUCTURES) $(OBJ_X11)
-lX11 $(HEADERS_SUPERLU_DIST) $(LIBS_SUPERLU_DIST)
  111:         @echo
  112:
  113:
  114:
  115: # BSLD Programs --------------------------------------------------------
  116: #
  117: bsld-%-partial:   OPTIONS += -DBSLD_PARTIAL_PIVOTING
  118: bsld-%-threshold: OPTIONS += -DBSLD_THRESHOLD_PIVOTING
  119:
  120: bsld-lin-partial:    $(DIR_BINARIES)/bsld-lin-partial
  121: bsld-lin-threshold:  $(DIR_BINARIES)/bsld-lin-threshold
```

```
  122: bsld-omp-partial:    $(DIR_BINARIES)/bsld-omp-partial
  123: bsld-omp-threshold:  $(DIR_BINARIES)/bsld-omp-threshold
  124: bsld-mpi-partial:    $(DIR_BINARIES)/bsld-mpi-partial
  125: bsld-mpi-threshold:  $(DIR_BINARIES)/bsld-mpi-threshold
  126:
  127: $(DIR_BINARIES)/bsld-lin-%: main-bsld.cpp bsld.h $(OBJ_BSLD)
  128:         @echo \# 'basename $@'
  129:         $(CPP) $(OPTIONS) -o $@ $< $(OBJ_BSLD) -lX11 $(HEADERS_RB) $(LIBS_RB
)  $(LIB_CBLAS)
  130:         @echo
  131:
  132: $(DIR_BINARIES)/bsld-mpi-%: main-bsld.cpp bsld.h $(OBJ_BSLD)
  133:         @echo \# 'basename $@'
  134:         @echo . /opt/gcdistro/modules/Modules/init/bash
  135:         @echo module purge
  136:         @echo module load mpi/openmpi
  137:         $(MPI_CPP) -D_MPI -fopenmp $(OPTIONS) -o $@ $< $(OBJ_BSLD) -lX11 $(
HEADERS_RB) $(LIBS_RB)  $(LIB_CBLAS)
  138:         @echo
  139:
  140: $(DIR_BINARIES)/bsld-omp-%: main-bsld.cpp bsld.h $(OBJ_BSLD)
  141:         @echo \# 'basename $@'
  142:         $(CPP) -fopenmp $(OPTIONS) -o $@ $< $(OBJ_BSLD) -lX11 $(HEADERS_RB)
$(LIBS_RB)  $(LIB_CBLAS)
  143:         @echo
  144:
  145:
  146: # Utility Programs ------------------------------------------------
  147: #
  148: hund:       $(DIR_BINARIES)/hund
  149: rb2png:     $(DIR_BINARIES)/rb2png
  150: rbprint:    $(DIR_BINARIES)/rbprint
  151: rbls:       $(DIR_BINARIES)/rbls
  152: rbview:     $(DIR_BINARIES)/rbview
  153: rbconv:     $(DIR_BINARIES)/rbconv
  154:
  155:
  156: $(DIR_BINARIES)/hund: main-hund.cpp $(OBJ_DATASTRUCTURES) hund.h $(OBJ_X11)
  157:         @echo \# 'basename $@'
  158:         $(CPP) $(OPTIONS) -o $@ $< $(OBJ_DATASTRUCTURES) $(OBJ_X11) -lX11 $(
HEADERS_HUND) $(LIBS_HUND)
  159:         @echo
  160:
  161: $(DIR_BINARIES)/rb2png: main-util-rb2png.cpp $(OBJ_DATASTRUCTURES) out-png.h
  162:         @echo \# 'basename $@'
  163:         $(CPP) $(OPTIONS) -o $@ $< $(OBJ_DATASTRUCTURES) -lpng $(HEADERS_RB)
$(LIBS_RB)
  164:         @echo
  165:
  166: $(DIR_BINARIES)/rbprint: main-util-rbprint.cpp $(OBJ_DATASTRUCTURES)
  167:         @echo \# 'basename $@'
  168:         $(CPP) $(OPTIONS) -o $@ $< $(OBJ_DATASTRUCTURES) $(HEADERS_RB) $(
LIBS_RB)
  169:         @echo
  170:
  171: $(DIR_BINARIES)/rbls: main-util-rbls.cpp $(OBJ_DATASTRUCTURES)
  172:         @echo \# 'basename $@'
  173:         $(CPP) $(OPTIONS) -o $@ $< $(OBJ_DATASTRUCTURES) $(HEADERS_RB) $(
LIBS_RB)
  174:         @echo
  175:
  176: $(DIR_BINARIES)/rbview: main-util-rbview.cpp $(OBJ_DATASTRUCTURES) $(OBJ_X11)
  177:         @echo \# 'basename $@'
```

```
  178:          $(CPP) $(OPTIONS) -o $@ $< $(OBJ_DATASTRUCTURES) $(OBJ_X11) -lX11 $(
HEADERS_RB) $(LIBS_RB)
  179:          @echo
  180:
  181: $(DIR_BINARIES)/rbconv: main-util-rbconv.cpp $(OBJ_DATASTRUCTURES)
  182:          @echo \# `basename $@`
  183:          $(CPP) $(OPTIONS) -o $@ $< $(OBJ_DATASTRUCTURES) $(HEADERS_RB) $(
LIBS_RB)
  184:          @echo
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    July 2010
    4:     File:    out-png.h
    5:     Purpose: This file contains the function definition for generating
    6:              PNG images from matrices.
    7: */
    8:
    9: #ifndef OUT_PNG_H
   10: #define OUT_PNG_H
   11:
   12: #include <cstdio>
   13: #include <png.h>
   14: #include "datastructures.h"
   15:
   16: inline void setRGB(png_byte *ptr, int val);
   17: int *createImage( int width, int height, const sparseMatrix &M, bool shade );
   18: int writeBuffer(char* filename, int width, int height, int *buffer, char*
title);
   19:
   20: int writeImage(char* filename, int width, int height, const sparseMatrix &M,
bool shade, char* title=NULL)
   21: {
   22:     int *buffer = NULL;
   23:     buffer = createImage(width, height, M, shade);
   24:     return writeBuffer(filename, width, height, buffer, title);
   25: }
   26:
   27: int writeBuffer(char* filename, int width, int height, int *buffer, char*
title)
   28: {
   29:     if (buffer == NULL) {
   30:         return 1;
   31:     }
   32:
   33:     int code = 0;
   34:     FILE *fp;
   35:     png_structp png_ptr;
   36:     png_infop info_ptr;
   37:     png_bytep row;
   38:
   39:     // Open file for writing (binary mode)
   40:     fp = fopen(filename, "wb");
   41:     if (fp == NULL) {
   42:         fprintf(stderr, "Could not open file %s for writing\n", filename);
   43:         code = 1;
   44:         goto finalise;
   45:     }
   46:
   47:     // Initialize write structure
   48:     png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL,
NULL);
   49:     if (png_ptr == NULL) {
   50:         fprintf(stderr, "Could not allocate write struct\n");
   51:         code = 1;
   52:         goto finalise;
   53:     }
   54:
   55:     // Initialize info structure
   56:     info_ptr = png_create_info_struct(png_ptr);
   57:     if (info_ptr == NULL) {
   58:         fprintf(stderr, "Could not allocate info struct\n");
   59:         code = 1;
```

```
    60:         goto finalise;
    61:     }
    62:
    63:     // Setup Exception handling
    64:     if (setjmp(png_jmpbuf(png_ptr))) {
    65:         fprintf(stderr, "Error during png creation\n");
    66:         code = 1;
    67:         goto finalise;
    68:     }
    69:
    70:     png_init_io(png_ptr, fp);
    71:
    72:     // Write header (8 bit colour depth)
    73:     png_set_IHDR(png_ptr, info_ptr, width, height,
    74:             8, PNG_COLOR_TYPE_RGBA, PNG_INTERLACE_NONE,
    75:             PNG_COMPRESSION_TYPE_BASE, PNG_FILTER_TYPE_BASE);
    76:
    77:     // Set title
    78:     if (title != NULL) {
    79:         png_text title_text;
    80:         title_text.compression = PNG_TEXT_COMPRESSION_NONE;
    81:         title_text.key = (char*)"Title";
    82:         title_text.text = title;
    83:         png_set_text(png_ptr, info_ptr, &title_text, 1);
    84:     }
    85:
    86:     png_write_info(png_ptr, info_ptr);
    87:
    88:     // Allocate memory for one row (3 bytes per pixel - RGB)
    89:     // Allocate memory for one row (4 bytes per pixel - RGBA)
    90:     row = new png_byte[4*width];
    91:
    92:     // Write image data
    93:     int x, y;
    94:     for (y=0 ; y<height ; y++) {
    95:         for (x=0 ; x<width ; x++) {
    96:             setRGB( &(row[x*4]), buffer[y*width + x] );
    97:         }
    98:         png_write_row(png_ptr, row);
    99:     }
   100:
   101:     // End write
   102:     png_write_end(png_ptr, NULL);
   103:
   104:     finalise:
   105:     if (fp != NULL) fclose(fp);
   106:     if (info_ptr != NULL) png_free_data(png_ptr, info_ptr, PNG_FREE_ALL, -1);
   107:     if (png_ptr != NULL) png_destroy_write_struct(&png_ptr,
(png_infopp)NULL);
   108:     if (row != NULL) delete[] row;
   109:
   110:     delete[] buffer;
   111:
   112:     return code;
   113: }
   114:
   115: inline void setRGB(png_byte *ptr, int val)
   116: {
   117:     if( val>0 )
   118:     {
   119:         ptr[0] = 255-val;
   120:         ptr[1] = 255-val;
   121:         ptr[2] = 255;
```

```
122:         ptr[3] = 255;
123:     }
124:     else
125:     {
126:         ptr[0] = ptr[1] = ptr[2] = 255;
127:         ptr[3] = 255;
128:     }
129: }
130:
131: int *createImage( int width, int height, const sparseMatrix &M, bool shade )
132: {
133:     int *buffer = new int[width*height];
134:     if( buffer == NULL ){
135:         fprintf(stderr, "Could not create image buffer\n");
136:         return NULL;
137:     }
138:
139:     //Clear the image
140:     for( int i=0; i<width*height; i++ )
141:         buffer[i] = 0;
142:
143:
144:     denseMatrix pixelCount( width, height );
145:     int sum=0, count=0, min=width*height, max=0;
146:     double average=0;
147:     double variance=0;
148:     int levels = 0;
149:
150:     //Scale the matrix elements onto an image grid.
151:     //
152:     int row, col=0;
153:     int x, y;
154:     for( int i=0; i<M.nnz; i++ ) {
155:         row = M.adjncy[i];
156:         if( i == M.colptr[col+1] )
157:             col++;
158:         x = col/static_cast<double>(M.n)*static_cast<double>(width);
159:         y = row/static_cast<double>(M.m)*static_cast<double>(height);
160:
161:         if( width>M.n || height>M.m ){
162:             for( int ii=0; ii<height/M.m; ii++ )
163:             for( int jj=0; jj<width/M.n; jj++ )
164:                 pixelCount[y+ii][x+jj]++;
165:         }
166:         else
167:             pixelCount[y][x]++;
168:     }
169:
170:     if( !shade )
171:     {
172:         for( int x=0; x<width;  x++ )
173:         for( int y=0; y<height; y++ )
174:         {
175:             if( pixelCount[y][x]!=0 )
176:                 buffer[ y*width + x ] = 255;
177:         }
178:         return buffer;
179:     }
180:
181:     //Determine the average density of matrix elements
182:     //  in the image.
183:     //
184:     for( int x=0; x<width;  x++ )
```

```
  185:     for( int y=0; y<height; y++ )
  186:     {
  187:         if( pixelCount[y][x]!=0 )
  188:         {
  189:             if(pixelCount[y][x] > max)
  190:                 max = pixelCount[y][x];
  191:             if( pixelCount[y][x] < min )
  192:                 min = pixelCount[y][x];
  193:             count++;
  194:             sum += pixelCount[y][x];
  195:         }
  196:     }
  197:     average = sum/count;
  198:
  199:
  200:     //Determine the standard deviation of matrix elements
  201:     //  in the image.
  202:     //
  203:     for( int x=0; x<width;  x++ )
  204:     for( int y=0; y<height; y++ )
  205:     {
  206:         if( pixelCount[y][x]!=0 )
  207:         {
  208:             double difference;
  209:             difference = pixelCount[y][x] - average;
  210:             variance += (difference*difference);
  211:         }
  212:     }
  213:     variance = sqrt( variance/count );
  214:     levels = static_cast<int>(((max-min+1)/variance)+.5);
  215:
  216:
  217:     int level;
  218:     double scale;
  219:     for( int x=0; x<width;  x++ )
  220:     for( int y=0; y<height; y++ )
  221:     {
  222:         if(pixelCount[y][x]!=0)
  223:         {
  224:             level = static_cast<int>((max - pixelCount[y][x] + 1)/variance +
.5);
  225:
  226:             scale = (levels-level)/(double)levels;
  227:             scale = scale*.9+.1;
  228:
  229:             buffer[ y*width + x ] = static_cast<int>( scale * 255 + .5);
  230:         }
  231:     }
  232:
  233:
  234:     return buffer;
  235: }
  236:
  237: #endif
  238:
```

```
 1: /*
 2:     Author:  Esteban Torres
 3:     Date:    November 2012
 4:     File:    out-x11.cpp
 5:     Purpose: This file contains the implementation details of the functions
 6:              for displaying matrices in an X11 window.
 7:
 8:     Data Structures:
 9:         1. Dense Matrix is used to scale the matrix onto the image.
10:         2. Image vectors [x] [y] [v] are used to store points/values.
11:         3. Linked List is used to store set of matrices.
12:         4. Class is used to store individual matrix info.
13: */
14:
15: #include "out-x11.h"
16:
17: matrix::matrix( const matrix& B ):
18:     M( B.M),
19:     px(B.px),
20:     py(B.py),
21:     shade(B.shade),
22:     m(B.m),
23:     n(B.n),
24:     nnz(B.nnz),
25:     xyv(NULL)
26: {
27:     xyv = new int[nnz*3];
28:
29:     for( int i=0; i<nnz*3; i++ )
30:         xyv[i] = B.xyv[i];
31: }
32:
33: void matrix::operator=( const matrix& B )
34: {
35:     if( this==&B ) return;
36:     M = B.M;
37:     px = B.px;
38:     py = B.py;
39:     shade = B.shade;
40:     m = B.m;
41:     n = B.n;
42:     nnz = B.nnz;
43:
44:     if( xyv!=NULL )
45:         delete [] xyv;
46:
47:     xyv = new int[nnz*3];
48:
49:     for( int i=0; i<nnz*3; i++ )
50:         xyv[i] = B.xyv[i];
51: }
52: void matrix::clear()
53: {
54:     nnz = 0;
55:     if( xyv!=NULL )
56:         delete [] xyv;
57:     xyv = NULL;
58: }
59: void matrix::render(int w, int h)
60: {
61:     clear();
62:
63:     denseMatrix canvas(w,h);
```

```
    64:
    65:     int row=0, col=0;
    66:     int x, y;
    67:     for( int i=0; i<M.nnz; i++ )
    68:     {
    69:         row = M.adjncy[i];
    70:         if( i == M.colptr[col+1] )
    71:             col++;
    72:         x = col/static_cast<double>(M.n)*static_cast<double>( w );
    73:         y = row/static_cast<double>(M.m)*static_cast<double>( h );
    74:
    75:         if( w>M.n || h>M.m )
    76:         {
    77:             for( int ii=0; ii<h/M.m; ii++ )
    78:             for( int jj=0; jj<w/M.n; jj++ )
    79:                 canvas[y+ii][x+jj]++;
    80:
    81:         }
    82:         else
    83:             canvas[y][x]++;
    84:     }
    85:
    86:     nnz = 0;
    87:     for( int i=0; i<h; i++ )
    88:     for( int j=0; j<w; j++ )
    89:     {
    90:         if( canvas[i][j]!=0 )
    91:             nnz++;
    92:     }
    93:
    94:     xyv = new int[ nnz*3 ];
    95:
    96:     {
    97:         int k=0;
    98:         for( int i=0; i<h; i++ )
    99:         for( int j=0; j<w; j++ )
   100:         {
   101:             if( canvas[i][j]!=0 )
   102:             {
   103:                 xyv[ k*3   ] = j;
   104:                 xyv[ k*3+1 ] = i;
   105:                 xyv[ k*3+2 ] = canvas[i][j];
   106:                 k++;
   107:             }
   108:         }
   109:     }
   110:
   111:     if(!shade)
   112:     {
   113:         for( int i=0; i<nnz; i++ )
   114:             xyv[ i*3+2 ] = 255;
   115:         return;
   116:     }
   117:
   118:     // Standard Deviation Shading
   119:     //
   120:     //
   121:     int min=w*h, max=0, sum=0;
   122:     double average, variance=0, difference, scale;
   123:     int levels, level;
   124:
   125:     for( int i=0; i<nnz; i++ )
   126:     {
```

```
127:         if( xyv[ i*3+2 ] > max )
128:             max = xyv[ i*3+2 ];
129:         if( xyv[ i*3+2 ] < min )
130:             min = xyv[ i*3+2 ];
131:         sum+= xyv[ i*3+2 ];
132:     }
133:     average = sum/nnz;
134:
135:     for( int i=0; i<nnz; i++ )
136:     {
137:         difference = xyv[ i*3+2 ] - average;
138:         variance += (difference*difference);
139:     }
140:     variance = sqrt( variance/nnz );
141:     levels = static_cast<int>(((max-min+1)/variance)+.5);
142:
143:     for( int i=0; i<nnz; i++ )
144:     {
145:         level = static_cast<int>((max - xyv[ i*3+2 ] + 1)/variance + .5);
146:
147:         scale = (levels-level)/(double)levels;
148:         scale = scale*.9+.1;
149:
150:         xyv[ i*3+2 ] = static_cast<int>( scale * 255 + .5);
151:     }
152:
153:     //
154:     //
155:     //  End of Standard Deviation Shading
156: }
157: matrix::~matrix()
158: {
159:     if( xyv!=NULL )
160:         delete [] xyv;
161:     xyv = NULL;
162: }
163:
164: x11Window::x11Window( unsigned int size, char* title ):
165:     height(size),
166:     width(size),
167:     min_height(size),
168:     min_width(size),
169:     gridwidth(0),
170:     gridheight(0),
171:     objwidth( size ),
172:     objheight( size ),
173:     objpadding( .075*size ),
174:     drawBlocks(false),
175:     blockSize(0),
176:     current(0),
177:     shade(false),
178:     dsp(NULL),
179:     blues(NULL)
180: {
181:     dsp = XOpenDisplay( NULL );
182:     if(dsp==NULL)
183:         return;
184:
185:     screenNumber = DefaultScreen(dsp);
186:
187:     win = XCreateSimpleWindow(dsp,
188:                               DefaultRootWindow(dsp),
189:                               200, 200,   // origin
```

```
190:                                width, height, // size
191:                                0, BlackPixel(dsp,screenNumber), // border
192:                                WhitePixel(dsp,screenNumber) );  // background
193:
194:     XStoreName(dsp,win,title);
195:
196:     gc = XCreateGC( dsp, win, 0, NULL );
197:     colorMap = XDefaultColormap( dsp, screenNumber );
198:     XParseColor( dsp, colorMap, "#C0C0C0", &gray );
199:     XAllocColor( dsp, colorMap, &gray );
200:
201:     char* color = new char[10];
202:     blues = new XColor[256];
203:     for( int i=0; i<=255; i++ )
204:     {
205:         sprintf( color, "#%02X%02X%02X", 255-i, 255-i, 255 );
206:         XParseColor( dsp, colorMap, color, blues+i );
207:         XAllocColor( dsp, colorMap, blues+i );
208:     }
209:     delete [] color;
210:
211:
212:     // register interest in the delete window message
213:     //
214:     Atom wmDeleteMessage = XInternAtom(dsp, "WM_DELETE_WINDOW", False);
215:     XSetWMProtocols(dsp, win, &wmDeleteMessage, 1);
216: }
217:
218: x11Window::x11Window( unsigned int size, vector<string> &files, bool s ):
219:     height(size),
220:     width(size),
221:     min_height(size),
222:     min_width(size),
223:     gridwidth(0),
224:     gridheight(0),
225:     objwidth( size ),
226:     objheight( size ),
227:     objpadding( .075*size ),
228:     drawBlocks(false),
229:     blockSize(0),
230:     filenames(files),
231:     current(0),
232:     shade(s),
233:     dsp(NULL),
234:     blues(NULL)
235: {
236:     dsp = XOpenDisplay( NULL );
237:     if(dsp==NULL)
238:         return;
239:
240:     screenNumber = DefaultScreen(dsp);
241:
242:     win = XCreateSimpleWindow(dsp,
243:                                DefaultRootWindow(dsp),
244:                                200, 200,   // origin
245:                                width, height, // size
246:                                0, BlackPixel(dsp,screenNumber), // border
247:                                WhitePixel(dsp,screenNumber) );  // background
248:
249:     XStoreName(dsp,win,filenames[current].c_str() );
250:
251:     gc = XCreateGC( dsp, win, 0, NULL );
252:     colorMap = XDefaultColormap( dsp, screenNumber );
```

```
   253:     XParseColor( dsp, colorMap, "#C0C0C0", &gray );
   254:     XAllocColor( dsp, colorMap, &gray );
   255:
   256:     char* color = new char[10];
   257:     blues = new XColor[256];
   258:     for( int i=0; i<=255; i++ )
   259:     {
   260:         sprintf( color, "#%02X%02X%02X", 255-i, 255-i, 255 );
   261:         XParseColor( dsp, colorMap, color, blues+i );
   262:         XAllocColor( dsp, colorMap, blues+i );
   263:     }
   264:     delete [] color;
   265:
   266:
   267:     // register interest in the delete window message
   268:     //
   269:     Atom wmDeleteMessage = XInternAtom(dsp, "WM_DELETE_WINDOW", False);
   270:     XSetWMProtocols(dsp, win, &wmDeleteMessage, 1);
   271:
   272:     openFile();
   273: }
   274:
   275: x11Window::~x11Window()
   276: {
   277:     if( blues!=NULL )
   278:         delete[] blues;
   279: }
   280:
   281: void x11Window::addMatrix( const sparseMatrix& MM, unsigned int posy,
unsigned int posx, bool shade, int b )
   282: {
   283:     if(dsp==NULL)
   284:         return;
   285:
   286:     list<matrix>::iterator itr = matrices.begin();
   287:
   288:     for( ;itr!= matrices.end(); itr++ )
   289:     {
   290:         if( itr->px == posx && itr->py == posy )
   291:             abort();
   292:     }
   293:
   294:     list<matrix>::iterator mtx;
   295:     matrices.push_back( matrix(MM, posy, posx, shade) );
   296:     mtx = matrices.end();
   297:     mtx--;
   298:     mtx->render( objwidth, objheight );
   299:
   300:     if( objheight > static_cast<unsigned int>(mtx->m) && b>0 )
   301:     {
   302:         drawBlocks = true;
   303:         blockSize = static_cast<double>(objheight)/mtx->m*b;
   304:     }
   305:
   306:     //Resize the window to fit the matrix
   307:     //
   308:     if( posy >= gridheight )
   309:         gridheight = posy+1;
   310:     if( posx >= gridwidth )
   311:         gridwidth = posx+1;
   312:
   313:     width = (gridwidth*objwidth) + (gridwidth+1)*objpadding;
   314:     height= (gridheight*objheight) + (gridheight+1)*objpadding;
```

```
  315:
  316:     min_width  = gridwidth;
  317:     min_height = gridheight;
  318:
  319:     XResizeWindow(dsp,win,width,height);
  320: }
  321:
  322: void x11Window::draw()
  323: {
  324:     if(dsp==NULL)
  325:         return;
  326:
  327:     list<matrix>::iterator itr = matrices.begin();
  328:
  329:     for( ;itr!= matrices.end(); itr++ )
  330:     {
  331:         for( int i=0; i<itr->nnz; i++ )
  332:         {
  333:             XSetForeground( dsp, gc, blues[ itr->xyv[i*3+2] ].pixel );
  334:             XDrawPoint(dsp, win, gc,
  335:                 itr->xyv[i*3]   + itr->px*objwidth  + (itr->px+1)*objpadding,
  336:                 itr->xyv[i*3+1] + itr->py*objheight + (itr->py+1)*objpadding
  337:             );
  338:         }
  339:
  340:         XSetForeground( dsp, gc, gray.pixel );
  341:
  342:         XDrawRectangle( dsp, win, gc,
  343:                         itr->px*objwidth +(itr->px+1)*objpadding-2,
  344:                         itr->py*objheight+(itr->py+1)*objpadding-2,
  345:                         objwidth+2,
  346:                         objheight+2 );
  347:         if( drawBlocks )
  348:         {
  349:             for( double i=blockSize; i<static_cast<double>(objheight);
i+=blockSize )
  350:             {
  351:                 XDrawLine( dsp, win, gc,
  352:                     itr->px*objwidth +(itr->px+1)*objpadding-2,
  353:                     itr->py*objheight+(itr->py+1)*objpadding + static_cast<
int>(i+.05),
  354:                     itr->px*objwidth +(itr->px+1)*objpadding-2 + objwidth+2,
  355:                     itr->py*objheight+(itr->py+1)*objpadding + static_cast<
int>(i+.05) );
  356:                 XDrawLine( dsp, win, gc,
  357:                     itr->px*objwidth +(itr->px+1)*objpadding + static_cast<
int>(i+.05),
  358:                     itr->py*objheight+(itr->py+1)*objpadding-2,
  359:                     itr->px*objwidth +(itr->px+1)*objpadding + static_cast<
int>(i+.05),
  360:                     itr->py*objheight+(itr->py+1)*objpadding-2 + objheight+2
);
  361:             }
  362:         }
  363:     }
  364: }
  365:
  366: void x11Window::resize(unsigned int new_width, unsigned int new_height)
  367: {
  368:     XClearWindow( dsp, win );
  369:
  370:     double width_ratio  = objwidth/(double)width;
  371:     double height_ratio = objheight/(double)height;
```

```
372:     double padd_ratio   = objpadding/(double)width;
373:
374:     objwidth = width_ratio * new_width;
375:     objheight = height_ratio * new_height;
376:
377:     width = new_width;
378:     height = new_height;
379:
380:     objpadding = padd_ratio*width;
381:
382:     list<matrix>::iterator itr = matrices.begin();
383:
384:     for( ;itr!= matrices.end(); itr++ )
385:         itr->render( objwidth, objheight );
386:
387:     draw();
388: }
389:
390: void x11Window::openFile()
391: {
392:     sparseMatrix A;
393:     char* thefile;
394:     int stringlen = filenames[current].length()+1;
395:
396:     thefile = new char[ stringlen ];
397:     memcpy( thefile, filenames[current].c_str(), stringlen );
398:
399:     if( A.rbRead( thefile )!= 0 )
400:     {
401:         fprintf(stderr, "Cannot open %s\n", thefile );
402:         delete [] thefile;
403:         return;
404:     }
405:     XStoreName(dsp,win,filenames[current].c_str() );
406:
407:     delete [] thefile;
408:     matrices.clear();
409:     addMatrix(A,0,0,shade);
410: }
411:
412: void x11Window::nextFile()
413: {
414:     if( filenames.size()!=0 &&
415:         static_cast<unsigned int>(current+1) < filenames.size() )
416:     {
417:         current++;
418:         openFile();
419:         XClearWindow( dsp, win );
420:         draw();
421:     }
422: }
423:
424: void x11Window::prevFile()
425: {
426:     if( static_cast<unsigned int>(current) != 0 )
427:     {
428:         current--;
429:         openFile();
430:         XClearWindow( dsp, win );
431:         draw();
432:     }
433: }
434:
```

```
  435: void x11Window::show()
  436: {
  437:     // Resize variables
  438:     //
  439:     Window       root_return;
  440:     int          x_return;
  441:     int          y_return;
  442:     unsigned int width_return;
  443:     unsigned int height_return;
  444:     unsigned int border_width_return;
  445:     unsigned int depth_return;
  446:     unsigned int width_new;
  447:     unsigned int height_new;
  448:     //
  449:     //-----------------
  450:
  451:     if(dsp==NULL)
  452:         return;
  453:     XMapWindow( dsp, win );
  454:
  455:     XSelectInput( dsp, win, StructureNotifyMask );
  456:
  457:     for(;;)
  458:     {
  459:         XNextEvent( dsp, &evt );
  460:
  461:         if( evt.type == MapNotify  )
  462:         {
  463:             draw();
  464:             break;
  465:         }
  466:     }
  467:
  468:     XSelectInput( dsp, win, ExposureMask | StructureNotifyMask |
KeyPressMask );
  469:
  470:     for(;;) // Event Loop
  471:     {
  472:         XNextEvent( dsp, &evt );
  473:
  474:         if( evt.type == KeyPress )
  475:         {
  476:             switch( XLookupKeysym(&evt.xkey, 0) )
  477:             {
  478:                 case XK_Left:  prevFile(); break;
  479:                 case XK_Right: nextFile(); break;
  480:                 case XK_q:
  481:                 case XK_Q: goto OUT; break;
  482:                 case XK_S:
  483:                 case XK_s:  list<matrix>::iterator itr = matrices.begin();
  484:                             for( ;itr!= matrices.end(); itr++ )
  485:                             {
  486:                                 itr->shade = !itr->shade;
  487:                                 itr->render( objwidth, objheight );
  488:                             }
  489:
  490:                             XClearWindow( dsp, win );
  491:                             draw();
  492:                             break;
  493:             }
  494:         }
  495:         else if( evt.type == ConfigureNotify )
  496:         {
```

```
  497:             XGetGeometry( dsp, win,
  498:                           &root_return,
  499:                           &x_return, &y_return,
  500:                           &width_return, &height_return,
  501:                           &border_width_return, &depth_return
  502:                         );
  503:
  504:             width_new =width_return;
  505:             height_new=height_return;
  506:
  507:             if( width_new == width && height_new == height )
  508:                 continue;
  509:
  510:             if(width_new*min_height < height_new*min_width)
  511:                 height_new=(width_new*min_height)/min_width;
  512:             else if(width_new*min_height > height_new*min_width)
  513:                 width_new=(height_new*min_width)/min_height;
  514:
  515:             if((width_new <min_width ) || (height_new<min_height))
  516:             {
  517:                 width_new =min_width;
  518:                 height_new=min_height;
  519:             }
  520:
  521:             if((height_new!=height_return) || (width_new !=width_return ))
  522:             {
  523:                 width_new =width_new -width_new %min_width;
  524:                 height_new=(width_new*min_height)/min_width;
  525:
  526:                 XClearWindow( dsp, win );
  527:
  528:                 XResizeWindow(dsp,win,width_new,height_new);
  529:             }
  530:             else resize( width_new, height_new );
  531:         }
  532:         else if( evt.type == Expose )
  533:         {
  534:             draw();
  535:         }
  536:         else if( evt.type == DestroyNotify )
  537:         {
  538:             break;
  539:         }
  540:         else if( evt.xclient.data.l[0] == static_cast<long>(XInternAtom(dsp,
"WM_DELETE_WINDOW", False)) )
  541:         {
  542:             break;
  543:         }
  544:     }
  545:     OUT: ;
  546:     XDestroyWindow( dsp, win );
  547:     XCloseDisplay( dsp );
  548: }
```

```
 1: /*
 2:     Author:  Esteban Torres
 3:     Date:    September 2011
 4:     File:    out-x11.h
 5:     Purpose: This file contains the function definition for displaying
 6:              matrices in an X11 window.
 7:
 8:     Data Structures:
 9:         1. Dense Matrix is used to scale the matrix onto the image.
10:         2. Image vectors [x] [y] [v] are used to store points/values.
11:         3. Linked List is used to store set of matrices.
12:         4. Class is used to store individual matrix info.
13: */
14:
15: #ifndef OUT_X11_H
16: #define OUT_X11_H
17:
18: #include <X11/Xlib.h>
19: #include <X11/Xutil.h>
20: #include <X11/Xos.h>
21: #include "datastructures.h"
22: #include "blockmatrix.h"
23: #include <list>
24: #include <vector>
25: #include <string>
26:
27: class matrix
28: {
29:     public:
30:         sparseMatrix M;
31:         unsigned int px;
32:         unsigned int py;
33:         bool shade;
34:         int m;
35:         int n;
36:         int nnz;
37:         int* xyv;
38:
39:         matrix(const sparseMatrix& MM, int ppy, int ppx, bool s):
40:             M(MM),
41:             px(ppx),
42:             py(ppy),
43:             shade(s),
44:             m(0),
45:             n(0),
46:             nnz(0),
47:             xyv(NULL){};
48:         matrix( const matrix& B );
49:         void operator=(const matrix& B);
50:         void clear();
51:         void render(int w, int h);
52:         ~matrix();
53: };
54:
55: class x11Window
56: {
57:     private:
58:
59:         unsigned int width;
60:         unsigned int height;
61:         unsigned int min_width;
62:         unsigned int min_height;
63:
```

```
    64:         unsigned int gridwidth;
    65:         unsigned int gridheight;
    66:
    67:         unsigned int objwidth;
    68:         unsigned int objheight;
    69:         unsigned int objpadding;
    70:
    71:         bool drawBlocks;
    72:         double blockSize;
    73:
    74:         list<matrix> matrices; //List of matrices in window
    75:
    76:         vector<string> filenames; //List of matrix filenames
    77:         int current; //Current Matrix
    78:         bool shade;
    79:
    80:         //--------------------
    81:         //
    82:         Display *dsp;
    83:         int screenNumber;
    84:         Window win;
    85:         XEvent evt;
    86:         GC gc;
    87:         Colormap colorMap;
    88:         XColor* blues;
    89:         XColor gray;
    90:         //--------------------
    91:
    92:         void draw();
    93:         void resize(unsigned int new_width, unsigned int new_height);
    94:         void openFile();
    95:         void nextFile();
    96:         void prevFile();
    97:
    98:     public:
    99:         x11Window( unsigned int size, char* title=(char*)"");
   100:         x11Window( unsigned int size, vector<string> &files, bool s );
   101:         ~x11Window();
   102:
   103:         void addMatrix( const sparseMatrix& MM, unsigned int posy, unsigned
int posx, bool shade, int b=0 );
   104:         void show();
   105: };
   106:
   107: #endif
```

README.txt, Page 1 of 1

```
    1: This program requires the following libraries:
    2:     PaToH:       http://bmi.osu.edu/~umit/software.html
    3:     SuperLU:     http://crd.lbl.gov/~xiaoye/SuperLU/
    4:                  http://www.cs.berkeley.edu/~demmel/SuperLU.html
    5:     SuiteSparse: http://www.cise.ufl.edu/research/sparse/SuiteSparse/
    6:
```

```
 1: /*
 2:     Author: Esteban Torres
 3:     Date: April 2012
 4:     File: timer.h
 5:     Purpose: Define a timer class.
 6: */
 7:
 8: #ifndef _MY_TIMER_H_
 9: #define _MY_TIMER_H_
10:
11: #ifdef _OPENMP
12: #include <omp.h>
13: #else
14: #include <ctime>
15: #endif
16:
17: #define TIME(TIMER,PROCESS) TIMER.start();\
18:                             PROCESS;\
19:                             TIMER.stop();
20:
21: class timer
22: {
23:     private:
24: #ifdef _OPENMP
25:         double clock1, clock2;
26: #else
27:         clock_t clock1, clock2;
28: #endif
29:         double runTime,totalTime;
30:     public:
31:         timer():
32:             clock1(0),
33:             clock2(0),
34:             runTime(0),
35:             totalTime(0)
36:             {};
37:         void start();
38:         void stop();
39:         double getTotalTime();
40:         double getRunTime();
41: };
42:
43: void timer::start()
44: {
45: #ifdef _OPENMP
46:     clock1 = omp_get_wtime();
47: #else
48:     clock1 = clock();
49: #endif
50:     runTime = 0;
51: }
52:
53: void timer::stop()
54: {
55: #ifdef _OPENMP
56:     clock2 = omp_get_wtime();
57:     runTime = clock2 - clock1;
58:     totalTime += runTime;
59: #else
60:     clock2 = clock();
61:     runTime = static_cast<double>(clock2-clock1)/CLOCKS_PER_SEC;
62:     totalTime += runTime;
63: #endif
```

```
64: }
65:
66: double timer::getTotalTime()
67: {
68:     return totalTime;
69: }
70:
71: double timer::getRunTime()
72: {
73:     return runTime;
74: }
75:
76: #endif
```

```
    1: /*
    2:     Author:  Esteban Torres
    3:     Date:    August 2012
    4:     File:    verify.h
    5:     Purpose: This file is used to verify L and U factors
    6:              by solving a simple equation.
    7: */
    8:
    9: #ifndef VERIFY_H
   10: #define VERIFY_H
   11:
   12: #include <cmath>
   13: #include <limits>
   14:
   15: #include "datastructures.h"
   16:
   17: class solveStats
   18: {
   19:     public:
   20:         double acc; //Accuracy; Tolerance .2
   21:         int co;     //Correct
   22:         int in;     //Incorrect
   23:
   24:         solveStats():
   25:             acc(0),co(0),in(0)
   26:             {};
   27: };
   28:
   29: void lu_solve( CSFBlock &csrL, CSFBlock &csrU, denseMatrix &B, denseMatrix
&X );
   30:
   31:
   32: void verify( sparseMatrix &A, sparseMatrix &L, sparseMatrix &U, solveStats
&stats )
   33: {
   34:     int m,n;
   35:
   36:
   37:     m = A.m;
   38:     n = A.n;
   39:
   40:
   41:     denseMatrix B(m,1);
   42:     denseMatrix X(m,1);
   43:
   44:
   45:
   46:     //1. Create initial X vector [1..2...3..]
   47:     //
   48:     for( int i=0; i<m; i++ )
   49:         X[i][0] = i;
   50:
   51:     //2. Multiply A*X to get B
   52:     //
   53:     B = A*X;
   54:
   55:     //Initialize X
   56:     //
   57:     for( int i=0; i<m; i++ )
   58:         X[i][0] = 0;
   59:
   60:
   61:     //Convert L to CSR
```

```
    //
    CSFBlock csrL( m, n );
    csrL = L;
    csrL = csrL.toCSR();

    //Convert U to CSR
    //
    CSFBlock csrU( m, n );
    csrU = U;
    csrU = csrU.toCSR();



    lu_solve( csrL, csrU, B, X );


    //Iterative Refinement
    //

    denseMatrix  R( m, 1 );
    denseMatrix AX( m, 1 );

    int itr=0;
    double berr=0,lastberr=100;
    iterate:
    {
        // 1. Multiply A*X
        // 2. Compute Residual
        // 3. Solve new equation
        // Repeat

        //Initialize residual
        for( int i=0; i<m; i++ )
            R[i][0]  = 0;


        //1. Multiply A*X
        //
        AX = A*X;

        //2. Compute Residual
        //
        for( int i=0; i<m; i++ )
            R[i][0] = B[i][0] - AX[i][0];

        lu_solve( csrL, csrU, R, AX ); //Solve

        for( int i=0; i<m; i++ )
            berr = max( berr, abs(R[i][0])/(abs(AX[i][0])+abs(B[i][0])) );

        if( berr > std::numeric_limits<double>::epsilon() && berr <=
.5*lastberr )
        {

            for( int i=0; i<m; i++ )
                X[i][0] += AX[i][0];
            lastberr = berr;
            berr = 0;
            itr++;
            goto iterate;
        }
    }

```

```
  124:
  125:     for( int i=0; i<m; i++ )
  126:     {
  127:         if( abs( X[i][0]-i ) < 0.2 )
  128:             stats.co++;
  129:     }
  130:
  131:     stats.in = m - stats.co;
  132:
  133:     stats.acc = stats.co/(double)m*100;
  134: }
  135:
  136:
  137: void lu_solve( CSFBlock &csrL, CSFBlock &csrU, denseMatrix &B, denseMatrix
&X )
  138: {
  139:     int m, n;
  140:     m = csrL.m;
  141:     n = csrL.n;
  142:     double *Y = NULL;
  143:
  144:     Y = new double[ m ];
  145:     assert( Y!=NULL );
  146:
  147:
  148:     //Initialize Y
  149:     //
  150:     for( int i=0; i<m; i++ )
  151:         Y[i] = 0;
  152:
  153:
  154:     //Solve A*X=B == L*U*X = L*Y = B for X.
  155:     //Given A, L, U, B
  156:     //  1. Solve L*Y=B for Y
  157:     //  2. Solve U*X=Y for X
  158:     //
  159:
  160:     // Solve for Y
  161:     {
  162:         int j=0, row=0, col=0;
  163:         double product=0;
  164:         for( int i=0; i<(int)csrL.V.size(); i++ )
  165:         {
  166:             col = csrL.I[i];
  167:             if( i==csrL.P[j] )
  168:             {
  169:                 row = j;
  170:                 product = 0;
  171:                 while(csrL.P[++j]<0);
  172:             }
  173:             if( row==col )
  174:                 Y[row] = B[row][0]-product;
  175:             else
  176:                 product += csrL.V[i]*Y[col];
  177:         }
  178:     }
  179:
  180:     //Solve for X
  181:     {
  182:         int j=0, row=0, col=0;
  183:         double product=0;
  184:         j=m;
  185:         while( csrU.P[--j]<0 );
```

```
186:          row = j;
187:
188:          for( int i=csrU.nnz()-1; i>=0; i-- )
189:          {
190:              col = csrU.I[i];
191:
192:              if( i==csrU.P[j] )
193:              {
194:                  X[row][0] = (Y[row] - product) / csrU.V[i];
195:
196:                  while( csrU.P[--j]<0 );
197:                  row = j;
198:                  product = 0;
199:              }
200:              else
201:                  product += csrU.V[i]*X[col][0];
202:          }
203:      }
204:
205:
206:      delete [] Y;
207: }
208:
209:
210:
211: #endif
```

BIOGRAPHICAL SKETCH

Esteban Torres graduated from The Science Academy of South Texas in May 2004. He attended The University of Texas-Pan American and earned a Bachelor's of Science in Computer Engineering in 2009. He continued his studies at The University of Texas-Pan American, working closely with his thesis advisor Dr. Yul Chu. As a teaching assistant, he helped administer labs, developed lab course work, and helped with grading for the computer engineering and computer science departments. Later, he joined Dr. Chu as a research assistant, and developed a distributed sparse direct solver. During this time, he was also involved with the installation, documentation, and testing of a new high-performance PC cluster. In August 2013, he earned a Master's of Science in Electrical Engineering for his work on distributed sparse direct solvers.

For correspondence, please write to: Esteban Torres, 3511 Daytona Avenue, McAllen, Texas 78503 or email etorres2@gmail.com.