User Guide for NICSLU

Xiaoming Chen

chenxm05@mails.tsinghua.edu.cn

Nano-scale Integrated Circuit and System (NICS) Laboratory

Tsinghua National Laboratory for Information Science and Technology

Department of Electronic Engineering, Tsinghua University

August 27, 2013

Contents

1 License **3**

2 Introduction **3**

3 Matrix Format **4**

4 Using NICSLU in a C/C++ Program **5**

4.1 Data Types Used in NICSLU 5

4.2 The `SNicsLU` Structure 6

4.2.1 Readable Members 6

4.2.2 Writable Members 7

4.3 Function Return Values 8

4.4 NICSLU Routines 9

4.4.1 `NicsLU_Initialize` 9

4.4.2 `NicsLU_Destroy` 9

4.4.3 `NicsLU_CreateMatrix` 10

4.4.4 `NicsLU_CreateThreads` 10

4.4.5 `NicsLU_DestroyThreads` 10

4.4.6 `NicsLU_BindThreads` 10

4.4.7 `NicsLU_CreateScheduler` 11

4.4.8 `NicsLU_Analyze` 11

4.4.9 `NicsLU_Factorize` 11

4.4.10 `NicsLU_ReFactorize` 11

4.4.11 `NicsLU_Factorize_MT` 12

4.4.12 `NicsLU_ReFactorize_MT` 12

4.4.13 `NicsLU_Solve` 12

4.4.14 NicsLU_SolveFast 12
4.4.15 NicsLU_ResetMatrixValues 12
4.4.16 NicsLU_Residual 12
4.4.17 NicsLU_Refine 13
4.4.18 NicsLU_Throughput 13
4.4.19 NicsLU_Flops 13
4.4.20 NicsLU_ThreadLoad 14
4.4.21 NicsLU_Transpose 14
4.4.22 NicsLU_DumpA 14
4.4.23 NicsLU_DumpLU 15
4.4.24 NicsLU_ConditionNumber 15
4.4.25 NicsLU_MemoryUsage 16
4.4.26 NicsLU_Sort 16
4.4.27 NicsLU_MergeDuplicateEntries 16

5 Complex Number Package NICSLUc 16

6 Compilation and Test 17

6.1 System Requirements 17
6.2 Folders and Files 17
6.3 Compilation 17
6.3.1 Compilation on Windows 17
6.3.2 Compilation on Linux 18
6.4 Test Demo Programs 18
6.5 Link NICSLU to Your Programs 18
6.6 Remarks 18

7 History 19

1 License

2 Introduction

NICSLU is a high-performance and robust software package for solving large-scale sparse linear systems of equations ($Ax = b$) on shared-memory machines. It is written by C, and can be easily used in C/C++ programs.

NICSLU solves $Ax = b$ by Gaussian elimination method (LU factorization). It factorizes matrix A into product of a lower triangular matrix L and an upper triangular matrix U (i.e. $A = LU$, numerical factorization step), and then the solution of $Ax = b$ is obtained by solving two triangular equations $Ly = b$ and $Ux = y$ (right-hand-solving step). Matrix A doesn't need to be symmetric or definite, but it must be square and full-rank, otherwise NICSLU cannot solve it.

Generally speaking, a simple description of sparse Gaussian elimination is as follows. Matrix A is factorized to:

$$LM_{n-1}R_{n-1}\cdots M_1R_1 = PD_rAD_cQ$$

where n is the dimension of A; D_r and D_c are two diagonal matrices to scale A to enhance numerical stability; P and Q are row and column permutation matrices, which are used to maintain sparsity (i.e. reduce fill-ins); R_k is the column permutation matrix generated by partial pivoting that occurs at step k during numerical factorization; M_k is an upper triangular matrix whose kth row contains the multipliers. So $Ax = b$ can be solved by:

$$\begin{aligned} & x = A^{-1}b \\ & = \left(D_r^{-1}P^{-1}LM_{n-1}R_{n-1}\cdots M_1R_1Q^{-1}D_c^{-1}\right)^{-1}b \\ & = D_cQR_1^{-1}M_1^{-1}\cdots R_{n-1}^{-1}M_{n-1}^{-1}L^{-1}PD_rb \end{aligned}$$

NICSLU is based on the sparse left-looking algorithm proposed by Gilbert and Peierls [1], and KLU algorithm proposed by Davis [2]. We use a more efficient static pivoting algorithm (HSL_MC64 algorithm) [3,4], which is combined with partial pivoting to achieve higher numerical stability. We have developed a novel parallel algorithm, which obtains effective acceleration on shared-memory multi-core processors [5–7].

There are also some other similar software packages, such as SuperLU [8–10], PARDISO [11], etc. NICSLU is different from these software packages because NICSLU does not utilize the BLAS. NICSLU is well suited for extremely sparse matrices, such as matrices in circuit simulation problems. In addition, NICSLU specially supports the case that requires many factorizations with the same nonzero pattern but different values.

NICSLU can be obtained from http://nicslu.weebly.com.

If you are using NICSLU in your research, please cite the following three papers:

[1] Xiaoming Chen, Wei Wu, Yu Wang, Hao Yu, Huazhong Yang, "An EScheduler-based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation", Circuits and Systems II: Express Briefs, IEEE Transactions on, vol. 58, no. 10, pp. 702-706, oct. 2011.

[2] Xiaoming Chen, Yu Wang, Huazhong Yang, "NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation", Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 32, no. 2, pp. 261-274, feb. 2013.

[3] Xiaoming Chen, Yu Wang, Huazhong Yang, "An Adaptive LU Factorization Algorithm for Parallel Circuit Simulation", Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific, pp.359-364, Jan. 30, 2012-Feb. 2, 2012.

3 Matrix Format

$$\begin{pmatrix} 1.1 & 0 & 0 & -7.7 & 13.13 & 0 \\ 0 & 2.2 & 0 & 0 & 9.9 & 0 \\ 0 & 8.8 & -3.3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4.4 & 0 & 0 \\ 0 & 0 & 11.11 & 0 & 5.5 & 0 \\ 10.1 & 0 & 0 & 12.12 & 0 & 6.6 \end{pmatrix}$$

Ax	1.1	-7.7	13.13	2.2	9.9	8.8	-3.3	-4.4	11.11	5.5	10.1	12.12	6.6
Ai	0	3	4	1	4	1	2	3	2	4	0	3	5

Ap	0	3	5	7	8	10	13

Figure 1: Example to illustrate the CSR format.

NICSLU uses the compressed sparse row (CSR) format to store a sparse matrix, as illustrated in Fig. 1 illustrates. CSR uses five parameters to describe a sparse matrix, as listed in the below.

- `n`: (unsigned) integer, matrix dimension, i.e. the matrix is `n`×`n`. NICSLU only supports square matrices.
- `nnz`: (unsigned) integer, the number of nonzeros in the matrix.
- `Ai`: (unsigned) integer array of length `nnz`, storing the column indices of all nonzeros.
- `Ax`: floating-point array of length `nnz`, storing the values of all nonzeros.
- `Ap`: (unsigned) integer array of length `n+1`, storing the location of the first nonzeros of each row in `Ai` and `Ax`. The first and last elements must be `Ap[0]=0` and `Ap[n]=nnz`. Values of the ith row of the matrix are stored in `Ax[Ap[i]], Ax[Ap[i]+1],` $\cdots$`, Ax[Ap[i+1]-1]`, and the corresponding column indices of the nonzeros are stored in `Ai[Ap[i]], Ai[Ap[i]+1],` $\cdots$`, Ai[Ap[i+1]-1]`, number of nonzeros of the ith row is `Ap[i+1]-Ap[i]`. The matrix is zero-based stored, which means the row and column indices are in the range from `0` to `n-1`.

The transposed format of CSR is compressed sparse column (CSC), which is stored in column-major.

4 Using NICSLU in a C/C++ Program

4.1 Data Types Used in NICSLU

NICSLU uses several self-defined data types, as listed in Table 1, in which the first column lists the data types used in NICSLU, and the second column lists the corresponding data types in standard C. The detailed definitions of the data types can be found in `nics_config.h`.

Table 1: Data types used in NICSLU.

data type	**C type**	**meaning**
`int__t`	`int` or `long long`	32-bit or 64-bit[a] integer
`uint__t`	`unsigned int` or `unsigned long long`	32-bit or 64-bit[a] unsigned integer
`real__t`	`double`	double-precision floating-point
`bool__t`	`unsigned char`	boolean value: `TRUE` or `FALSE`
`size_t/size__t`	`size_t`	32-bit or 64-bit[b] unsigned long integer
`byte__t`	`unsigned char`	byte, 8-bit

[a] According to whether the macro `NICS_INT64` is defined.

[b] According to the hardware platform and the compiling configurations.

4.2 The SNicsLU Structure

The sole `SNicsLU` structure in NICSLU contains all configurations, matrix data, LU factors, and statistical information for LU factorization. This object appears in most NICSLU functions as the first parameter. Details of `SNicsLU` are given in `nicslu.h`. Only a few member parameters of `SNicsLU` can be read or written by users, which are listed below, users should not change the other parameters.

4.2.1 Readable Members

All the members in floating-point array `stat` are readable, and the meanings of each indexed member is as follows.

- `real__t stat[0]`: analysis time, runtime (in seconds) of `NicsLU_Analyze`.
- `real__t stat[1]`: factorization time, runtime (in seconds) of `NicsLU_Factorize` or `NicsLU_Factorize_MT` (according to your last calling).
- `real__t stat[2]`: re-factorization time, runtime (in seconds) of `NicsLU_ReFactorize` or `NicsLU_ReFactorize_MT` (according to your last calling).
- `real__t stat[3]`: right-hand-solving time, runtime (in seconds) of `NicsLU_Solve` or `NicsLU_SolveFast` (according to your last calling).
- `real__t stat[4]`: initialization time of the scheduler, runtime (in seconds) of `NicsLU_CreateScheduler`.
- `real__t stat[5]`: total number of floating-point operations (FLOPs) to factorize the matrix, which is generated by `NicsLU_Flops`.
- `real__t stat[6]`: condition number of the matrix, which is estimated by `NicsLU_ConditionNumber`. If MC64 scaling is used, the condition number is estimated after MC64 scaling to the matrix.
- `real__t stat[7]`: estimated speedup if all the cores of the CPU are used, which is calculated by `NicsLU_CreateScheduler`.
- `real__t stat[8]`: estimated upper bound of speedup attainable by NICSLU, regardless of the number of cores, which is calculated by `NicsLU_CreateScheduler`.
- `real__t stat[9]`: number of cores on the computer. If super-threading is supported and enabled, `stat[9]` is twice of the number of physical cores.
- `real__t stat[10]`: estimated number of FLOPs to factorize the matrix, which is calculated by `NicsLU_CreateScheduler`.
- `real__t stat[11]`: estimated number of nonzeros in $L + U - I$, which is calculated by `NicsLU_CreateScheduler`.

- `real__t stat[12]`: estimated memory throughput (in bytes), which is calculated by `NicsLU_Throughput`.
- `real__t stat[13]`: a suggestion. Non-zero suggests using `NicsLU_Factorize` function and zero suggests using `NicsLU_Factorize_MT`. The suggestion is generated by `NicsLU_CreateScheduler`.
- `real__t stat[14]`: number of off-diagonal pivots.
- `real__t stat[15]`: refinement time, runtime (in seconds) of `NicsLU_Refine`.
- `real__t stat[16]`: number of iterations in the refine process `NicsLU_Refine`.
- `real__t stat[21]`: memory usage (in bytes), which is calculated by `NicsLU_MemoryUsage`.

Besides the above members in `stat` array, the following members are also readable.

- `size_t l_nnz, u_nnz`: the two members indicate the number of nonzeros in L and U after factorization, including the diagonals of L and U respectively.
- `size_t lu_nnz`: number of nonzeros in $L + U - I$ after factorization, which is equal to `l_nnz + u_nnz - n`.

4.2.2 Writable Members

All the writable members are in unsigned integer array `cfgi` and floating-point array `cfgf`.

- `uint__t cfgi[0]`: default value is 0. A flag to indicate the CSR or CSC mode. Zero indicates CSR and non-zero indicates CSC. If your matrix is stored in CSC format, NICSLU can also directly deal with it. In this case, NICSLU solves $A^T x = b$.
- `uint__t cfgi[1]`: default value is 1. A flag to indicate whether using the MC64 algorithm to scale the matrix before factorization. MC64 scaling is strongly recommended.
- `uint__t cfgi[2]`: default value is 0. A flag to indicate the scaling method when factorizing the matrix. 1 indicates max-scaling, 2 indicates sum-scaling and other values indicate no scaling. Based on our experiments, the scaling methods may have effect in frequency-domain simulation, but they generally have no effect in time-domain transient simulation.
- `uint__t cfgi[3]`: default value is 16. It is a scheduling threshold for parallel LU factorization. It should be larger than or equal to the number of threads.
- `uint__t cfgi[4]`: default value is 2. It is used to pre-allocate memory for parallel LU factorization. If it is larger, NICSLU will use more memory, but during parallel LU factorization, less memory re-allocation will happen.

- `uint_t cfgi[7]`: default value is the number of created threads. This number indicates the actual number of threads used for parallel computation. For example, you can create 8 threads and only use 6 of them to perform parallel factorization. You can set this parameter before `NicsLU_Factorize_MT` or `NicsLU_ReFactorize_MT`. It cannot exceed the number of created threads.

- `real_t cfgf[0]`: default value is 0.001. It is the partial pivoting tolerance which should be less than 1.0. If the diagonal entry has a magnitude greater than or equal to `cfgf[0]` times the largest magnitude of entries in the pivot row, then the diagonal entry is selected as the pivot, otherwise an off-diagonal pivot will be chosen. If this parameter is larger, more off-diagonal pivots will be generated.

- `real_t cfgf[1]`: default value is 3.0. It is also used to pre-allocate memory for parallel LU factorization. If it is larger, NICSLU will use more memory, but during parallel LU factorization, less memory re-allocation will happen.

- `real_t cfgf[4]`: default value is 0.95. It is used to control the load balance for `NicsLU_Factorize_MT` and `NicsLU_ReFactorize_MT`. It should be around 1.0.

- `real_t cfgf[5]`: default value is 1.5. It is used to control the memory re-allocation growth. It should be larger than 1.0.

If not necessary, it is recommended that these configurations (writable members) keep the default values.

4.3 Function Return Values

Each NICSLU function returns an integer (`int`) to indicate whether the function is executed successfully or not. The return values and their meanings are listed in the below. You should check the return value of each function to avoid failures of NICSLU. Negative values indicate fatal failures and positive values indicate warnings generated.

- `NICS_OK`: value 0. The function is executed successfully.

- `NICSLU_GENERAL_FAIL`: value -1. A simple failure has occurred.

- `NICSLU_ARGUMENT_ERROR`: value -2. There are some errors with the function arguments; for example, you specify `NULL` to a pointer that is not allowed to be `NULL`.

- `NICSLU_MEMORY_OVERFLOW`: value -3. No enough memory.

- `NICSLU_FILE_CANNOT_OPEN`: value -4. The specified file cannot be opened.

- `NICSLU_MATRIX_STRUCTURAL_SINGULAR`: value -5. The matrix is structural singular, i.e. the matrix is not structural full-rank.

- `NICSLU_MATRIX_NUMERIC_SINGULAR`: value -6. The matrix is numerical singular, i.e. there is one row/column that does not contain any nonzero elements.

- `NICSLU_MATRIX_INVALID`: value -7. The matrix is invalid because there are some errors in the CSR/CSC storage. For example, an index is out of range.

- `NICSLU_MATRIX_ENTRY_DUPLICATED`: value -8. The matrix has duplicated entries in the CSR/CSC storage.

- `NICSLU_THREADS_NOT_INITIALIZED`: value -9. The threads are not created yet.

- `NICSLU_MATRIX_NOT_INITIALIZED`: value -10. The matrix is not created yet.

- `NICSLU_SCHEDULER_NOT_INITIALIZED`: value -11. The scheduler is not created yet.

- `NICSLU_SINGLE_THREAD`: value -12. When creating only 1 thread, this error occurs, since the main thread does not require to be explicitly created.

- `NICSLU_THREADS_INIT_FAIL`: value -13. The specified threads cannot be created.

- `NICSLU_MATRIX_NOT_ANALYZED`: value -14. The matrix is not analyzed yet.

- `NICSLU_MATRIX_NOT_FACTORIZED`: value -15. The matrix is not factorized yet.

- `NICSLU_NUMERIC_OVERFLOW`: value -16. Numerical overflow has occurred during factorization.

- `NICSLU_USE_SEQUENTIAL_FACTORIZATION`: value +1. It is returned by `NicsLU_CreateScheduler`, indicating sequential `NicsLU_Factorize` should be used rather than parallel `NicsLU_Factorize_MT`.

- `NICSLU_BIND_THREADS_FAIL`: value +2. The threads cannot be pined to cores.

4.4 NICSLU Routines

4.4.1 `NicsLU_Initialize`

```
int NicsLU_Initialize(SNicsLU *nicslu);
```

This function initializes the `SNicsLU` structure and sets the default configurations. It must be called first, before any other NICSLU function called. It should be called only once, otherwise memory leak will occur.

4.4.2 `NicsLU_Destroy`

```
int NicsLU_Destroy(SNicsLU *nicslu);
```

This function destroys the `SNicsLU` structure and frees all the memory allocated by NICSLU. It must be called at last, otherwise memory leak will occur. Repeatedly calling this function has no effect.

4.4.3 `NicsLU_CreateMatrix`

`int NicsLU_CreateMatrix(SNicsLU *nicslu, uint__t n, uint__t nnz, real__t *ax, uint__t *ai, uint__t *ap);`

This function initializes the matrix which will be used by NICSLU. The matrix is described by the CSR/CSC format (i.e. `n, nnz, ax, ai, ap`), which is described in Section 3. If your matrix is stored in CSC format, you can also directly use it, and after calling this function, `nicslu->cfgi[0]` should be set to a non-zero value.

This function resets all configurations to their default values. If you need to change the configurations, you should set them after calling this function.

If this function is repeatedly called, it first destroys the existing matrix and then creates the new matrix.

4.4.4 `NicsLU_CreateThreads`

`int NicsLU_CreateThreads(SNicsLU *nicslu, unsigned int thread, bool__t check);`

This function creates threads for parallel computation. The second argument (`thread`) specifies the number of threads, including the main thread. The last argument (`check`) specifies whether to check the number of threads or not. If it is `TRUE`, then this function will check your specified thread number, and if the thread number is larger than the number of cores on your computer, the thread number will be set to the core number.

We strongly recommend `check = TRUE`.

If you only want to run single-threaded LU factorization (i.e. sequential factorization), this function is not required, you should directly call the sequential version of factorization and re-factorization functions.

If this function is repeatedly called, it first destroys the existing threads and then creates the new threads.

The created threads will not exit until `NicsLU_DestroyThreads` or `NicsLU_Destroy` is called.

4.4.5 `NicsLU_DestroyThreads`

`int NicsLU_DestroyThreads(SNicsLU *nicslu);`

This function destroys the threads and frees memory used by the threads. It is contained in `NicsLU_Destroy`, so it can be skipped when you finish your computation.

Repeatedly calling this function has no effect.

4.4.6 `NicsLU_BindThreads`

`int NicsLU_BindThreads(SNicsLU *nicslu, bool__t unbind);`

This function binds threads to cores (`unbind = FALSE`) or unbinds threads from cores (`unbind = TRUE`). Binding threads to cores may increase the performance when the number of threads is much less than the number of cores because it avoids context switches.

However, when the number of threads is near or equal to the number of cores, binding threads to cores may lead to performance degradation.

It should be called after `NicsLU_CreateThreads`.

4.4.7 `NicsLU_CreateScheduler`

`int NicsLU_CreateScheduler(SNicsLU *nicslu);`

This function creates the task scheduler for parallel LU factorization. If you want to run parallel factorization or parallel re-factorization, it should be called after `NicsLU_Analyze`.

If this function returns `NICSLU_USE_SEQUENTIAL_FACTORIZATION` (value +1), it indicates that the matrix is not suitable for parallel factorization (i.e. the parallel performance may be even worse then the sequential performance, for the specified matrix). It returns `NICS_OK` (value 0) if the matrix is suitable for parallel factorization. So we suggest you choose the proper factorization function according to the return value of this function. Note: the suggestion is only for factorization but not re-factorization. `NicsLU_ReFactorize_MT` can always achieve speedups than `NicsLU_ReFactorize`.

The suggestion can also be obtained by `nicslu->stat[13]`.

If this function is repeatedly called, it first destroys the existing scheduler and then creates the new scheduler.

4.4.8 `NicsLU_Analyze`

`int NicsLU_Analyze(SNicsLU *nicslu);`

This function analyzes the matrix, including row/column ordering and MC64 scaling. It must be called after `NicsLU_CreateMatrix` and before any factorization or refactorization.

Repeatedly calling this function has no effect.

4.4.9 `NicsLU_Factorize`

`int NicsLU_Factorize(SNicsLU *nicslu);`

This function performs the numerical LU factorization (i.e. $A = LU$) with partial pivoting. It must be called after `NicsLU_Analyze`.

4.4.10 `NicsLU_ReFactorize`

`int NicsLU_ReFactorize(SNicsLU *nicslu, real__t *ax);`

If you want to factorize another matrix with different entry values but with the same nonzero structure, this function can be used. This function is without partial pivoting, so it uses the same pivoting order as the last `NicsLU_Factorize` or `NicsLU_Factorize_MT` called. It must be called after `NicsLU_Factorize` or `NicsLU_Factorize_MT` is called at least once. This function executes faster than `NicsLU_Factorize`; however, it may cause numerical stability problem. Array `ax` specifies the new matrix values in CSR/CSC format.

4.4.11 `NicsLU_Factorize_MT`

`int NicsLU_Factorize_MT(SNicsLU *nicslu);`

It is the parallel version of `NicsLU_Factorize`. `NicsLU_CreateScheduler` and `NicsLU_CreateThreads` should be called before this function.

4.4.12 `NicsLU_ReFactorize_MT`

`int NicsLU_ReFactorize_MT(SNicsLU *nicslu, real__t *ax);`

It is the parallel version of `NicsLU_ReFactorize`. `NicsLU_CreateScheduler` and `NicsLU_CreateThreads` should be called before this function.

4.4.13 `NicsLU_Solve`

`int NicsLU_Solve(SNicsLU *nicslu, real__t *rhs);`

This function performs right-hand-solving (i.e. $Ly = b$ and $Ux = y$) to obtain the solution of $Ax = b$. It can be called after any factorization or re-factorization functions.

Array `rhs` is used for both input and output. On input, it should store the right-hand-vector (b); on output, it is overwritten by the solution vector (x).

4.4.14 `NicsLU_SolveFast`

`int NicsLU_SolveFast(SNicsLU *nicslu, real__t *rhs);`

It is a faster version of `NicsLU_Solve`. When there are many zeros in the right-hand-vector (b), this function may be faster than `NicsLU_Solve`.

4.4.15 `NicsLU_ResetMatrixValues`

`int NicsLU_ResetMatrixValues(SNicsLU *nicslu, real__t *ax);`

Since `NicsLU_ReFactorize` and `NicsLU_ReFactorize_MT` are performed without partial pivoting, they may cause numerical stability problem. If you want to factorize a new matrix with the same nonzero pattern, and with partial pivoting to avoid the potential numerical stability problem, then this function should be used to reset the matrix data. And then `NicsLU_Factorize` or `NicsLU_Factorize_MT` can be used to factorize the new matrix with partial pivoting. Array `ax` specifies the new matrix values in CSR/CSC format.

4.4.16 `NicsLU_Residual`

`int NicsLU_Residual(uint__t n, real__t *ax, uint__t *ai, uint__t *ap, real__t *x, real__t *b, real__t *error, int norm, int mode);`

This function calculates the residual error of $||Ax - b||$.

ax, ai and ap are the CSR/CSC storage of matrix A. Array x is the solution vector and b is the right-hand-vector, both are inputs. norm indicates the norm of the residual: 1 indicates the 1-norm, 2 indicates the 2-norm and other values indicate the infinite-norm. mode indicates the CSR/CSC mode: zero indicates CSR and non-zero indicates CSC. On output, *error returns the residual error. error cannot be a NULL pointer.

4.4.17 NicsLU_Refine

`int NicsLU_Refine(SNicsLU *nicslu, real__t *x, real__t *b, real__t eps, uint__t maxiter);`

When necessary, this function can be used to refine the solution. However, it is not always successful. The refinement is implemented as follows:

```
compute residual r = Ax - b;
while ||r|| > eps
    solve Ad = r;
    update solution x = x - d;
    update residual r = Ax - b;
end while
```

The residual is based on the 1-norm. Array x should be the solution vector on input; on output, it will be updated by the refinement. Array b is the right-hand-vector (input). eps is the precision, when the residual is smaller than eps, the refinement ends. maxiter is used to control the refinement iterations. If maxiter is nonzero, the refinement will end when the number of iterations reaches maxiter; otherwise the number of iterations has no limit, but it will also end when the residual reaches a minimum value.

4.4.18 NicsLU_Throughput

`int NicsLU_Throughput(SNicsLU *nicslu, real__t *thr);`

This function estimates the memory throughput (in bytes), i.e. total amount of memory accesses that are required to factorize the matrix. Parameter *thr returns the throughput if thr is not NULL. It is an estimation of the throughput, the actual memory throughput may not be equal to the estimated value. The throughput can also be obtained by nicslu->stat[12].

4.4.19 NicsLU_Flops

`int NicsLU_Flops(SNicsLU *nicslu, real__t *flops);`

This function calculates the number of FLOPs that are required to factorize the matrix. Argument *flops returns the number of FLOPs if flops is not NULL. The number of FLOPs can also be obtained by nicslu->stat[5].

4.4.20 NicsLU_ThreadLoad

`int__t NicsLU_ThreadLoad(SNicsLU *nicslu, unsigned int threads, real__t **thread_flops);`

This function calculates the number of FLOPs of each thread such that one can evaluate load-balance of the parallel algorithm. Parameter `threads` specifies the number of threads, and the pointer `*thread_flops` must be `NULL`. On output, this function will allocate memory for `*thread_flops`, which is a floating-point array, with the length of the thread number. The number of FLOPs of thread No. `i` is stored in `(*thread_flops)[i]`. The thread number specified here may not equal to the actual thread number used or created in parallel factorization.

Example:

```
real__t *thread_flops;
thread_flops = NULL;
/*factorizing the matrix here ...*/
NicsLU_ThreadLoad(&nicslu, 8, &thread_load);
/*to obtain flops of thread i, visit thread_load[i]*/
free(thread_flops);
```

4.4.21 NicsLU_Transpose

`int NicsLU_Transpose(uint__t n, uint__t nnz, real__t *ax, uint__t *ai, uint__t *ap);`

This function transposes a matrix stored in CSR/CSC format. On input, you should specify `n, nnz, ax, ai, ap` to be the original matrix; on output, `ax, ai, ap` will be overwritten by the transposed matrix.

4.4.22 NicsLU_DumpA

`int NicsLU_DumpA(SNicsLU *nicslu, real__t **ax, uint__t **ai, uint__t **ap);`

This function stores matrix A into CSR format after factorization. The exported matrix is different from the original matrix since row/column ordering and MC64 scaling may be performed after analysis and factorization. Pointers `*ax, *ai, *ap` must be `NULL`, otherwise a memory exception or memory leak will occur. This function will allocate memory for these pointers.

Example:

```
real__t *ax;
uint__t *ai, *ap;
ax = NULL;
ai = NULL;
ap = NULL;
```

```
/*factorizing the matrix here ...*/
NicsLU_DumpA(nicslu, &ax, &ai, &ap);
/*do some processing ...*/
free(ax);
free(ai);
free(ap);
```

4.4.23 NicsLU_DumpLU

`int NicsLU_DumpLU(SNicsLU *nicslu, real__t **lx, uint__t **li, size_t **lp, real__t **ux, uint__t **ui, size_t **up);`

This function stores the factorized LU factors into CSR format. Pointers `*lx, *li, *lp, *ux, *ui, *up` must be `NULL`, otherwise a memory exception or memory leak will occur. This function will allocate memory for these pointers. The exported CSR arrays contain the diagonals of L and U. The number of nonzeros of L and U can be obtained from `nicslu->l_nnz` and `nicslu->u_nnz`.

Example:

```
real__t *lx, *ux;
uint__t *li, *ui;
size_t *lp, *up;
lx = ux = NULL;
li = ui = NULL;
lp = up = NULL;
/*factorizing the matrix here ...*/
NicsLU_DumpLU(nicslu, &lx, &li, &lp, &ux, &ui, &up);
/*do some processing ...*/
free(lx);
free(li);
free(lp);
free(ux);
free(ui);
free(up);
```

4.4.24 NicsLU_ConditionNumber

`int NicsLU_ConditionNumber(SNicsLU *nicslu, real__t *cond);`

This function estimates the condition number of the matrix, using the 1-norm. If MC64 scaling is used, the condition number is reported based on the scaled matrix, otherwise it's calculated based on the original matrix. Argument `*cond` returns the condition number if `cond` is not `NULL`. The condition number can also be obtained by `nicslu->stat[6]`.

4.4.25 NicsLU_MemoryUsage

`int NicsLU_MemoryUsage(SNicsLU *nicslu, real__t *memuse);`

This function estimates the memory used by NICSLU. `*memuse` will return the memory usage if `memuse` is not `NULL`. The memory usage can also be obtained by `nicslu->stat[21]`.

4.4.26 NicsLU_Sort

`int NicsLU_Sort(uint__t n, uint__t nnz, real__t *ax, uint__t *ai, uint__t *ap);`

This function sorts each row/column of a matrix stored in CSR/CSC format. On input, you should specify `n, nnz, ax, ai, ap` to be the original matrix; on output, `ax, ai, ap` will be overwritten by the sorted matrix.

4.4.27 NicsLU_MergeDuplicateEntries

`int NicsLU_MergeDuplicateEntries(uint__t n, uint__t *nnz, real__t **ax, uint__t **ai, uint__t **ap);`

This function merges duplicate entries of a matrix stored in CSR/CSC format. On input, you should specify `n, *nnz, *ax, *ai, *ap` to be the original matrix; on output, `*nnz` will be changed, and `*ax, *ai, *ap` will be re-allocated and overwritten by the new matrix.

5 Complex Number Package NICSLUc

NICSLUc is quite similar to the real number package NICSLU. Complex number is defined as `complex__t` in NICSLUc:

```
typedef struct __tag_complex
{
    real__t real;
    real__t image;
} complex__t;
```

The main data structure is `SNicsLUc`. All the routines in NICSLUc are with the prefix `NicsLUc_` instead of `NicsLU_`. Except for `NicsLU_ConditionNumber`, other routines have corresponding complex number routines.

The MC64 package in NICSLUc is different from that in NICSLU. It calculates the permutation arrays and the scaling factors based on the following matrix:

$$C = \begin{pmatrix} |c_{0,0}| & |c_{0,1}| & \cdots & |c_{0,n-1}| \\ |c_{1,0}| & |c_{1,1}| & \cdots & |c_{1,n-1}| \\ \vdots & \vdots & \ddots & \vdots \\ |c_{n-1,0}| & |c_{n-1,1}| & \cdots & |c_{n-1,n-1}| \end{pmatrix}$$

where $|\cdot|$ is the modulus of complex number.

6 Compilation and Test

6.1 System Requirements

NICSLU can be executed on Intel x86 or AMD64 (x86-64) hardware platforms, both Windows and GNU Linux are supported. To compile NICSLU, **Microsoft Visual Studio 2005 or higher version** (for Windows)/**gcc** (for Linux) is required.

NICSLU uses the Windows API (for Windows)/pthread library (for Linux) to manage threads. NICSLU does NOT require BLAS, OpenMP, or some other libraries.

Unlike some other parallel packages, the number of threads used in NICSLU can be conveniently controlled by `NicsLU_CreateThreads` and `nicslu->cfgi[7]`, NO environment variable is required.

6.2 Folders and Files

The NICSLU package contains folders and files shown in Table 2.

Table 2: Folders and files

name	description
demo\	two samples to show how to use NICSLU
doc\	user guide (it's me!)
include\	header files of NICSLU
lib\	object files and nicslu.a will be generated here
source\	source files of NICSLU
util\	some useful code
win_vs2012\	Windows project for Visual Studio 2012
lesser.txt	the GNU LGPL license
Makefile	makefile
make.inc	configurations of makefile
readme.txt	a simple description of compilation and test

6.3 Compilation

6.3.1 Compilation on Windows

We have provided a VS2012 project in "<top>\win_vs2012\" directory ("<top>\" is the top directory of NICSLU). The project includes four sub-projects which will generate "nicslu.lib", "nicslu_util.lib", "demos.exe", and "demop.exe". **Open "nicslu.sln" and simply compile this project (press F7) can complete the whole compilation process.** "nicslu.lib" and "nicslu_util.lib" are generated in "<top>\win_vs2012\Release\" (x86 compilation) or "<top>\win_vs2012\x64\Release\" (x64 compilation) directory, and "demos.exe" and "demop.exe" are generated in "<top>\demo\" directory.

If you are not using Visual Studio 2012, please follow the three steps.

- Create an empty **static library** project, add all files in "<top>\source\" into the project, change optimization flags, and then compile it. A static library named "<project name>.lib" will be generated.
- If you want to test demo programs, you should also compile the codes in "<top>\util\". Just create another static library project and do the similar things. Please also add "<top>\include\" (change it to a proper relative path according to the location of your project) to "Additional Include Directories".
- To compile demo programs, create an empty **console** project, add "<top>\demo\demos.c" or "<top>\demo\demop.p" (only one file) into the project. Also add "<top>\include\" and "<top>\util\" to "Additional Include Directories", and add the two libraries (.lib files) generated by the above two steps to "Additional Dependencies". Compile it.

6.3.2 Compilation on Linux

Just type "make" at the top directory. It will generate "nicslu.a" in "<top>/lib/", "nicslu_util.a" in "<top>/util/", and "demos" and "demop" in "<top>/demo/".

Please note the optimization flag can be only `-O2` when using gcc, using `-O3` will generate segmentation fault.

6.4 Test Demo Programs

If all the above steps are successful, just run "demos" (no arguments) or "demop" (command: demop <#threads>) in "<top>\demo" to test the sequential or parallel demo programs. For example, on Linux, the commands can be "./demos" or "./demop 4" when "<top>/demo/" is the current work directory.

6.5 Link NICSLU to Your Programs

On Windows, add "nicslu.lib" to "Additional Dependencies" of your program, or add the code

```
#pragma comment(lib, "nicslu.lib")
```

to any position of your codes.

On Linux, link with "nicslu.a" (`-L. nicslu.a`), the POSIX real-time extension library (`-lrt`), the pthread library (`-lpthread`), and the math library (`-lm`).

6.6 Remarks

Three macros can be used to control the features of NICSLU: `SSE2`, `NICS_INT64`, and `NO_EXTENSION`.

SSE2 indicates whether SSE2 instructions are enabled. If SSE2 is disabled, the code is optimized by the compiler; otherwise the hand-optimized SSE2-enabled code is used.

When NICSLU is compiled into 64-bit library, the bitwidth of `int__t` and `uint__t` are determined by the macro `NICS_INT64`. If `NICS_INT64` is defined, they are 64-bit integers, otherwise they are 32-bit integers. Note that `NICS_INT64` can be only used on 64-bit architectures. This option does NOT affect the maximum number of nonzeros in LU factors that NICSLU can store, but affects the maximum number of nonzeros matrix in A. If the number of nonzeros in A exceeds 4294967295 (0xFFFFFFFF), please define `NICS_INT64` in "make.inc".

`NO_EXTENSION` is used to control the feature of thread binding, since this feature is a non-standard GNU extension for Linux (for Windows, this feature is always supported). If you cannot compile NICSLU successfully, please define `NO_EXTENSION` in "make.inc".

7 History

2013, Aug 27. Version 3.0.1

* fix a small bug.

2013, Aug 17.

The complex number version is released.

2013, Jun 13. Version 3.0

This is a major update. Many new features are added.

* bug fixes.

* SSE2 is supported.

* `NicsLU_BindThreads` is added, which is used to bind threads to cores to improve the performance.

* util code is added.

* `NicsLU_Sort` and `NicsLU_MergeDuplicateEntries` are added.

* 64-bit integer is supported.

2013, Apr 16. Version 2.0

* NICSLU is distributed under the GNU LGPL license.

2012, Dec 16. Version 1.2

* the framework of NICSLU has a few changes, current framework is the final one and will not be changed in future versions.

* add function: `NicsLU_MemoryUsage`.

* add function: `NicsLU_DumpA`.

* add function: `NicsLU_SolveFast`.

* `NicsLU_ResetMatrixData` is changed to `NicsLU_ResetMatrixValues`.

* `NicsLU_ResidualError` is changed to `NicsLU_Residual`, the arguments are also changed.

* memory usage optimization.

* demo programs are changed.

* bug fixes.

2011, Oct 19. Version 1.1

* add function: `NicsLU_Throughput`.

* add function: `NicsLU_ThreadLoad`.

* `NicsLU_CreateScheduler` doesn't need to be called after `NicsLU_CreateThreads` anymore.

* correct an error in `NicsLU_ResidualError`.

* some small improvements.

* some small bug fixes.

2011, Jul 20. Version 1.0

* the first version is released.

References

[1] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.*, 9(5):862–874, 1988.

[2] T. A. Davis and E. P. Natarajan. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Trans. Math. Softw.*, 37(3):36:1–36:17, September 2010.

[3] I. S. Duff and J. Koster. The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, July 1999.

[4] I. S. Duff and J. Koster. On Algorithms For Permuting Large Entries to the Diagonal of a Sparse Matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, July 2000.

[5] X. Chen, Y. Wang, and H. Yang. NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(2):261 –274, feb. 2013.

[6] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang. An EScheduler-Based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 58(10):702–706, oct. 2011.

[7] X. Chen, Y. Wang, and H. Yang. An adaptive LU factorization algorithm for parallel circuit simulation. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 359–364, 30 2012-feb. 2 2012.

[8] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755, May 1999.

[9] J. W. Demmel, J. R. Gilbert, and X. S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[10] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003.

[11] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.*, 20(3):475–487, April 2004.